National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

UNIVERSITY OF ALBERTA

# A Universal Tiler for Integrated Circuit Layout

By

Mr. Douglas J. Konrad

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical Engineering

Edmonton, Alberta

Spring 1992

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73128-1

Canada

UNIVERSITY OF ALBERTA

Release Form

Name of Author: Mr. Douglas J. Konrad

Title of Thesis: A Universal Tiler for Integrated Circuit Layout

Degree: Master of Science

Year this Degree Granted: 1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific purposes only.

The author reserves all other publication and other rights in association with the copyright in this thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Douglas J. Konrad

Mr. Douglas J. Konrad
10950 89 Avenue
Edmonton, Alberta

April 24, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *A Universal Tiler for Integrated Circuit Layout* submitted by Mr. Douglas J. Konrad in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

Dr. J. T. Mowchenko

Dr. N. G. Durdle

Dr. M. T. Ozsu

April 24, 1992

## Abstract

Automated layout is the use of computer-based tools to create integrated circuit designs from higher-level specifications. Tilers are one type of automated layout tool, which combine previously designed pieces of layout, called tiles, into a two dimensional array. These rectangular tiles are connected together by abutting their boundaries together, thus completing the electrical connections between adjacent tiles. *Amoeba* is a new type of tiler, which makes the electrical connections between tiles by connecting together terminals located on the tiles which the input explicitly indicates must be connected. It then compacts the array in two one dimensional passes. Because of this difference, tiles can be designed for minimum size, rather than an arbitrary height and width, as with previous tilers. The problem is cast as a linear program, and the simplex method is used to solve it. The approach embodied in *Amoeba* is promising as shown in its results — it is able to assemble large structures, such as arithmetic logic units, and it effectively compacts finer structures, such as gate matrix cells, varying the row and column spacing to minimize the size of the cell.

## Acknowledgement

I offer my thanks to all who encouraged me in my studies; in particular, I want to acknowledge the support of the following people: My supervisor, Jack Mowchenko, my parents, John and Irma Konrad, my sister, Heather Konrad, my brother, Gary Konrad, and my friend, Norman Jantz.

# Contents


**1 Introduction** **1**

1.1 Automated Layout . . . . . 1
1.2 *Amoeba* . . . . . 2
1.3 *Amoeba's* Capabilities . . . . . 3
1.4 Outline . . . . . 3

**2 Background** **5**

2.0.1 Standard Cell Placement and Routing . . . . . 5
2.0.2 Macro-Cell Placement with Module Generation . . . . 7
2.1 Module Generators . . . . . 8
2.1.1 Layout Languages . . . . . 8
2.1.2 Leaf-Cell Synthesizers . . . . . 10
2.1.3 Programmable Logic Array Generators . . . . . 11
2.1.4 Weinberger Array Generators . . . . . 13
2.1.5 Gate Matrix Layout Tools . . . . . 14
2.1.6 Tilers and Iterated Structure Generators . . . . . 15
2.2 Amoeba . . . . . 19
2.3 Summary . . . . . 20

**3 Overview of *Amoeba*** **22**

3.1 Overview of *Amoeba* . . . . . 22
3.2 Program Inputs and Outputs . . . . . 24
3.2.1 Array Structure File . . . . . 24
3.2.2 Module Directory . . . . . 26
3.2.3 Design Rule File . . . . . 28
3.2.4 Algae Output . . . . . 30

**4 Constraint Generation** **34**
4.1 Formulating Layout as Linear Programming . . . . . 34
4.1.1 Linear Programming in Standard Form . . . . . 34
4.1.2 Geometrical Interpretation of Linear Programming . . 35
4.1.3 Tiling as Linear Programming . . . . . 37
4.1.4 Solving Linear Programs . . . . . 38
The Simplex Method — The Data Structures . . . . . 39
The Simplex Method — The Algorithm . . . . . 41
Alternatives to Simplex . . . . . 41
4.2 Constraint Generation . . . . . 42
4.2.1 Interference Constraint Generation . . . . . 43
4.2.2 Wiring Constraint Generation . . . . . 45
4.3 Feature Selection . . . . . 45
4.3.1 Feature Table Data Structure . . . . . 48
4.3.2 Rigid Straight Mode . . . . . 50
4.3.3 Rigid Dogleg Mode . . . . . 54
4.3.4 Unconstrained Straight Mode . . . . . 60
4.3.5 Interval Grove Data Structure . . . . . 62
4.3.6 Unconstrained Dogleg Mode . . . . . 64

**5 Implementation and Results** **66**
5.1 Implementation . . . . . 66
5.1.1 Programming . . . . . 67
5.1.2 Testing . . . . . 67
5.1.3 Efficiency . . . . . 69
Pivoting . . . . . 69
Memory Allocation . . . . . 70
Constraint Filtering . . . . . 71
5.2 Results . . . . . 72
5.2.1 Gate Matrix Example . . . . . 72
5.2.2 First ALU Example . . . . . 75
5.2.3 Second ALU Example . . . . . 76
5.2.4 Discussion . . . . . 78

**6 Conclusions** **79**
6.1 Summary of Results . . . . . 79
6.2 Future Work . . . . . 80

**Bibliography** 82

**A Legend for Diagrams of Mask Layout** 86

**B Meta-syntax for File Formats** 88

**C Data** 91
C.1 Gate Matrix Example . . . . . 91
C.2 First ALU Example . . . . . 97
C.3 Second ALU Example . . . . . 100

# List of Tables

5.1 Constraint Filter Performance . . . . . 72
5.2 Results for Gate Matrix Example . . . . . 72
5.3 Results for First ALU Example . . . . . 76
5.4 Results for Second ALU Example . . . . . 77

# List of Figures


2.1 Example of a Standard Cell Layout . . . . . 6
2.2 Example of a PLA Design . . . . . 12
2.3 Weinberger Array implementing exclusive-or in NMOS . . . . 14
2.4 Example of a Gate Matrix mask design (after [LL80]) . . . . . 15
2.5 Using *quilt* to build a Weinberger array . . . . . 17
2.6 Input used by Tpla to generate the PLA in Figure 2.2 . . . . 18

3.1 Example of an Array Structure File . . . . . 26
3.2 Array Structure File Syntax . . . . . 27
3.3 Portions of a Design Rule File . . . . . 28
3.4 Design Rule File Syntax . . . . . 29
3.5 Subset of Algae Used by *Amoeba* . . . . . 31
3.6 Example of Algae Produced by *Amoeba* . . . . . 32

4.1 Example of a Polytope . . . . . 36
4.2 An Interference Constraint . . . . . 37
4.3 A Straight Wiring Constraint . . . . . 38
4.4 *Amoeba's* Cost Function . . . . . 39
4.5 Interference Constraint Generation Algorithm . . . . . 44
4.6 Feature Selection Algorithm . . . . . 47
4.7 Example of a Feature Table Data Structure . . . . . 49
4.8 First Pass Rigid Straight Mode Algorithms . . . . . 51
4.9 Second Pass Rigid Straight Mode Algorithms . . . . . 52
4.10 Doglegs and Bounding Boxes . . . . . 54
4.11 Algorithm for Including Doglegs in an Element's Bounding Boxes . . . . . 55
4.12 Constraint Pairs and Line of Sight Routing . . . . . 56
4.13 Preventing Interference with a Long Dogleg . . . . . 57
4.14 First Pass Rigid Dogleg Mode Algorithms . . . . . 58

4.15 Second Pass Rigid Dogleg Mode Algorithms . . . . . 59
4.16 Second Pass Unconstrained Straight Mode Algorithms . . . . 61
4.17 Interval List Data Structure . . . . . 62
4.18 Second Pass Unconstrained Dogleg Mode Algorithms . . . . . 65

5.1 Gate Matrix Half Adder Layout Produced by *Amoeba* in Unconstrained Straight Mode . . . . . 73
5.2 Gate Matrix Half Adder Layout Produced in Unconstrained Straight Mode, Stripped of its Wiring . . . . . 74
5.3 Gate Matrix Half Adder Layout Produced in Unconstrained Dogleg Mode . . . . . 75
5.4 Rigid Straight Adder-Subtractor . . . . . 77
5.5 Rigid Straight Adder-Multiplier . . . . . 78
5.6 Example of "Random" Element Placement by *Amoeba* . . . . 78

A.1 Legend for Diagrams of Mask Layout . . . . . 87

B.1 Backus-Naur Form of Backus-Naur Form . . . . . 89

C.1 Gate Matrix Example, X, Rigid, Straight . . . . . 93
C.2 Gate Matrix Example, Y, Rigid, Straight . . . . . 94
C.3 Gate Matrix Example, X, Unconstrained, Straight . . . . . 94
C.4 Gate Matrix Example, Y, Unconstrained, Straight . . . . . 94
C.5 Gate Matrix Example, X, Rigid, Dogleg . . . . . 95
C.6 Gate Matrix Example, Y, Rigid, Dogleg . . . . . 95
C.7 Gate Matrix Example, X, Unconstrained, Dogleg . . . . . 95
C.8 Gate Matrix Example, Y, Unconstrained, Dogleg . . . . . 96
C.9 First ALU Example, X, Rigid, Straight . . . . . 98
C.10 First ALU Example, Y, Rigid, Straight . . . . . 98
C.11 First ALU Example, X, Unconstrained, Straight . . . . . 98
C.12 First ALU Example, Y, Unconstrained, Straight . . . . . 98
C.13 First ALU Example, X, Rigid, Dogleg . . . . . 99
C.14 First ALU Example, Y, Rigid, Dogleg . . . . . 99
C.15 First ALU Example, X, Unconstrained, Dogleg . . . . . 99
C.16 First ALU Example, Y, Unconstrained, Dogleg . . . . . 99
C.17 Second ALU Example, X, Rigid, Straight . . . . . 101
C.18 Second ALU Example, Y, Rigid, Straight . . . . . 101
C.19 Second ALU Example, X, Unconstrained, Straight . . . . . 101
C.20 Second ALU Example, Y, Unconstrained, Straight . . . . . 102

C.21 Second ALU Example, X, Rigid, Dogleg . . . . . 102
C.22 Second ALU Example, Y, Rigid, Dogleg . . . . . 102
C.23 Second ALU Example, X, Unconstrained, Dogleg . . . . . 102
C.24 Second ALU Example, Y, Unconstrained, Dogleg . . . . . 102

# Chapter 1

# Introduction

*Amoeba* is a computer program which assembles arrays using a description of the array's structure, and a library of pre-designed pieces of integrated circuit layout. It minimizes the size of the arrays through the use of linear programming. The key and original component of this thesis is the solution of the problem of generating the constraints which govern the placement of the tiles of the array. *Amoeba* is more flexible than previous computer programs which have tried to accomplish the same goal.

## 1.1 Automated Layout

Integrated circuits (ICs, chips) are fabricated by depositing layers of conductive and nonconductive material on the surface of a semiconductor substrate, then doping and etching the substrate and the deposited material. Photographic masks control the photolithographic process used when fabricating the circuits. All the basic circuit features, transistors, diodes, capacitors, resistors and interconnections, are specified on the masks. Finally, *layout* is the task of determining the location of these features on the surface of the IC.

Layout becomes *automated* when computer-based design tools are used to create mask designs from higher-level specifications. The input to the design tools is frequently a schematic netlist, i.e., a list of the devices in a circuit with their interconnections. The components in a schematic can be either transistors, logic gates or higher-level macro-cells, and a number of different methods can be used to generate the layout. *Standard cell place-and-route*, and *macro-cell placement* with *module generation* are both commonly used.

Automated layout is chosen over manual layout for one reason: it saves design time, which, in turn, results in much lower development costs. It also shortens the time needed to get a product to market, thus reducing debt service costs and improving the competitiveness of the developer. However, the use of automated layout usually results in some additional costs. Not only are the integrated circuits produced from automated layout frequently more expensive to manufacture, but the circuit performance is also penalized.

## 1.2 *Amoeba*

One type of automated layout tool is the *tiler*. A tiler is a program that assembles a module by combining previously designed sections of layout into a two dimensional array. Each of these layout sections is called a *tile* and, since these tools typically operate by abutting rectangular tiles together, they have been named "tilers". When the tiles abut, so do their wires, making the required connections to adjoining tiles.

*Amoeba* is a new type of tiler. Rather than being restricted to making electrical connections by abutting tile edges, a designer using *Amoeba* explicitly specifies the interconnections between the tiles. Because of this, the tiles can be designed to minimum size, rather than having to be a standard height and width. As a consequence, *Amoeba* can minimize the size of each row and column, instead of enforcing the standard dimensions. This results in arrays which are smaller than those produced by previously developed tilers.

Furthermore, *Amoeba* is unique in that it permits the interconnections between tiles to be bent. This makes it possible for tiles to be moved aside when larger tiles require more space. Again, this increased flexibility can result in smaller modules and therefore, in smaller chips.

Indeed, it is the objective of *Amoeba* to minimize the size of the modules it produces. The size of a layout is determined by the minimum proximity allowed between various mask features, and this in turn is specified by the *design rules* for the fabrication technology being used. When these design rules are applied to a particular array, a series of constraints is produced which controls the positions of the component tiles.

These constraints are linear, as is the cost function which is the size of the module. As a result, the problem of minimizing the size of a tiled module subject to these constraints is solved by a *linear program*. The simplex method was chosen as the algorithm to solve this linear program, because it

is easily implemented and performs well, compared to other algorithms for solving linear programs.

## 1.3 *Amoeba's* Capabilities

*Amoeba* tiles and compacts the module in two passes. In the first pass, it determines the positions of the tiles with respect to one axis, and in the second pass, with respect to the other. The order of the two passes is specified by the module designer.

Certain types of modules are regular, and can be most effectively compacted if that regularity is kept. For these, *rigid compaction* can be used. Guidelines, which must not be crossed by the tiles, are placed between the rows and columns of the array to ensure that tiles from one row or column do not overlap adjacent rows or columns. Other types of modules may be more effectively compacted without guidelines; *unconstrained compaction* can be selected for them.

Two types of wiring are also available. The first, *straight routing*, requires *Amoeba* to keep all interconnections straight and parallel to either the $x$ or $y$ axis. The second, *dogleg routing*, allows the wires to be bent into a dogleg shape, with a single jog perpendicular the longitudinal axis of each wire.

Finally, *Amoeba* allows cells to be stacked on top of one another to form composite elements. In this manner, features can be added to a simple cell. For example, a transistor used for gate matrix layout may have contacts between the diffusion and metal layers added, thus increasing the flexibility of the basic transistor cell.

## 1.4 Outline

The next chapter contains the results of a survey of the literature of automated layout. It provides a context for this project, and presents the concepts and justification for *Amoeba*. Chapter 3 gives an overview of *Amoeba's* operation, summarizes *Amoeba's* capabilities and discusses its inputs and output. Chapter 4 presents layout as linear programming and reviews the simplex method. It discusses calculation of a constraint between two elements, and selection of those elements for which constraints are necessary. The results produced by *Amoeba* are given in Chapter 5, as well as a discussion of the

implementation of *Amoeba* which covers some of the relevant aspects of software engineering applied to this project. Finally, Chapter 6 summarizes the results and presents the conclusions of this research project.

# Chapter 2

# Background

This chapter discusses work which has been done by others on tilers and on related areas. It shows the context in which *Amoeba* was developed and explains the motivation for this research.

The literature survey begins with a review of automated layout and an examination of the design process into which *Amoeba* and related software fits. Module generators (including layout-language-based generators, leaf-cell synthesizers, programmable logic array generators, Weinberger array generators and gate matrix tools) are discussed in detail. Next, tilers, and their utility in implementing semi-regular arrays are considered. The chapter concludes with a discussion of *Amoeba* — both the motivation for its development and its advantages over other tilers.

In Chapter 1, the rationale for the study and use of automated layout was presented. *Standard cell place-and-route* and *macro-cell placement* with *module generation* were mentioned as two commonly used methods. They are described below as a starting point in this survey.

### 2.0.1 Standard Cell Placement and Routing

*Standard cell placement and routing* is one example of automated layout [KG85]. This technique involves connecting pre-designed subcircuits, called standard cells, to implement the desired circuit. Each standard cell is designed for a particular function, and its layout conforms to a standard height which is enforced for all cells in a standard cell library. The cells are placed side by side to form long rows. Terminals on the sides of the cells allow the cells to be connected together by running wires in the channels between the

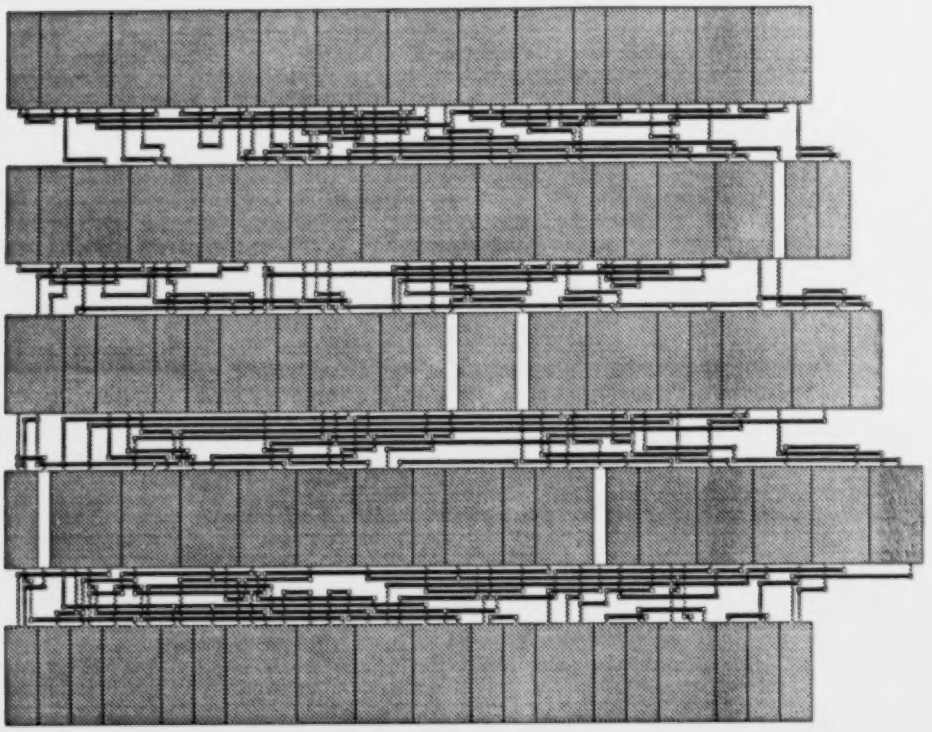


Figure 2.1: Example of a Standard Cell Layout

rows, as shown in Figure 2.1. The standard cell design procedure consists of the following steps. First, the functional specification for the integrated circuit is produced. Then the circuit is designed using the standard cells. Each cell implements a common function, such as a logic gate or a flip-flop (memory cell). The circuit design is used as input to a standard cell placement program which places the cells in stacks of rows. The placement program attempts to minimize some cost factor, such as total wiring length, chip area or the worst case delay of the signals between different cells. Finally, using the placement and the circuit schematic, a routing program routes the interconnecting wires through wiring channels left between the rows of cells.

Standard cell layout technology succeeds in removing the human designer from the layout phase of integrated circuit design. However, this success is not without a price. The area used for the wiring can approach 80% of the chip's area [KG85]. This will usually increase chip size substantially

as compared to those designed by fully custom methods, and this increase will reduce the yield of the fabrication process and thus increase production costs. Furthermore, the length of these wires increases the capacitive load on the cell outputs, thus reducing the speed of the circuit and increasing power requirements. This decreased speed can compromise the chip's ability to meet the performance objectives of the system for which it was designed. And the cost of the system will increase with a more power-hungry chip since a larger power supply and more cooling capacity must then be included. Frequently, however, the benefits of automated layout exceed these costs by a large margin, and many chips have been successfully and efficiently designed by using standard cell layout technology.

### 2.0.2 Macro-Cell Placement with Module Generation

An alternative to standard cell layout is *macro-cell placement* with *module generation*. This method follows "manual" design techniques more closely than does standard cell layout. In this method, the circuit is divided into functional blocks called macro-cells, or modules. A macro-cell differs from standard cells in that it typically performs a higher level function, and has an arbitrary rectangular shape. These macro-cells are placed on the chip in a placement called a *floorplan*. The modules can be designed using a number of different methods, depending on the function projected for the module. These methods include standard cell design, "manual" design, special function module generators and others that will be examined later in this chapter. Once the macro-cells have been generated and placed, they are connected to form the chip. The wires are assigned to the channels between the macro-cells in a phase called global routing. The detailed routing phase completes the layout by routing the individual wires within their assigned channels.

The hierarchical structure of a circuit schematic is retained when macro-cell placement is used. There are typically more connections between subordinate partitions within the same hierarchical grouping of a schematic than between subordinate partitions in different groupings. Because of this, keeping the subordinates of a group together will usually result in a smaller layout than occurs when the subordinates are placed anywhere. Hill compared layout (using standard cells) where the hierarchy is retained, and where it is ignored [Hil88]. When hierarchy is used, smaller and cleaner layouts are produced. In one example, use of hierarchy saved an area equivalent to 30,000 transistors in the control section of a microprocessor.

Macro-cell placement offers flexibility to the designer in that the floorplan can be "hand optimized." While this is theoretically possible for standard cells, their layout is extremely complex (often tens of thousands of cells) and the experience of a talented designer becomes useless in this sea of detail. Another advantage of the macro-cell design method is that each module can be synthesized using the method most appropriate to its function. Thus, methods such as standard cells, gate matrix layout or even "hand layout" can be selected as appropriate.

In the next sections, module generators will be examined in depth. These generators can all be used with the macro-cell design method just discussed. Certain types of module generators can benefit from *Amoeba*, as will be made clear further on in this chapter.

## 2.1 Module Generators

The term *module generator* will be used to describe any computer program which produces a layout of a module when given a specification of its structure or function. Effective and efficient module generators are essential if macro-cell placement is to be effective.

Certain module generators produce *irregular layouts*, which lack repetitive structures. Two types of irregular layout module generators are examined here: layout-language-based and leaf-cell synthesizers. Other module generators produce *semi-regular arrays*, which are VLSI structures that contain a partially repetitive pattern in one or two dimensions. Within an array the array elements may differ from position to position, but will be similarly sized. Generators for four types of arrays are discussed in this section: programmable logic arrays, Weinberger arrays, gate matrix layouts and iterated structures. A group of module generators (called tilers and iterated structure generators), closely related to semi-regular array generators, is also examined in this section.

### 2.1.1 Layout Languages

A layout-language-based module generator is a computer program which can only produce modules of a particular type. As a result, a layout-language-based module generator is usually restricted to a narrow problem, such as laying out an adder, given a set of parameters and constraints. For example,

a generator which produces registers may be given the width of a register, the desired current drive of its outputs and its speed, and then produce a layout which meets these criteria. A layout-language is a computer language which is able to describe geometric relationships between layout features. It may be a specifically designed language, or an existing language, extended to contain these operations. The quality of the layout produced by one of these tools depends on the skills of the person writing the module generator. Layout languages offer the flexibility and power of a programming language, but it is extremely difficult for a layout-language-based module generator to produce good layouts for general problems.

"i" is a layout language which is purely descriptive in nature [MBC83]. Geometric objects and their locations can be specified, but "i" does not contain any procedural operators. When objects are placed, their location can be described either as an absolute location, or relative to another object. Thus, if the first object is moved, the second will retain its placement relative to the first. This capability simplifies the task of the designer when specifying the location of subsections of a module. "ic" is an extension to C, allowing it to be used to produce "i" layouts. The procedural aspects of module generation are handled by a C program using "ic", and the basic geometric properties of the layout are handled in the "i" description.

Layla is similar to "i" and "ic" [Cor85]. It is a Pascal based layout-language targeted at parameterized cell libraries. A Layla program is translated into a Pascal program which is then compiled. When a cell is needed, the appropriate module generator is executed, and the cell parameters are provided as input. Layla supports hierarchy; it can place cells as well as basic geometric structures into a module.

ALI is a symbolic layout language [LNS+82]. Instead of specifying the positions of objects (either absolute or relative), the designer specifies that a given object is to be placed above, below, to the left of, or to the right of another object. ALI is programmed with knowledge of the target fabrication technology (the design rules), and will place the objects as close together as possible, while obeying the designer's relations. ALI is translated into Pascal, which is then compiled to produce the module generator.

ICEWATER is a specifically designed layout-language; it is not an extension of a general programming language [PE84]. Like Layla, it supports hierarchical construction. It also allows wiring to be completed by naming the two terminals to be connected, and permits arrays of components to be placed in the layout. If the facilities of ICEWATER are not powerful enough

for the designer, user-defined functions, in the language of the designer's choosing, are supported to supplement ICEWATER.

Tpack is a subroutine library which allows a designer to specify a tiled array [MO83]. It illustrates the use of a subroutine library as a layout language. Since tilers are examined in detail in section 2.1.6, Tpack will be discussed there.

### 2.1.2 Leaf-Cell Synthesizers

Leaf-cell synthesizers are a class of module generators which operate on schematics. A leaf-cell synthesizer should be able to take any transistor schematic and produce a good layout for the cell. Thus, instead of taking a parameterization of a particular function and producing a layout, the synthesizers will take a schematic of any function and produce a layout. In contrast, the layout languages can be used to create module generators which can produce a particular class of module. A chip designer might use such a generator to produce a six bit counter-adder which will operate in less than thirty nanoseconds. However, the same generator cannot produce a shift-register. A layout-language-based module generator is vertically integrated; it takes a high-level, functional description of a particular type of module and designs through several layers of abstraction to produce a layout. A leaf-cell synthesizer, however, is horizontally integrated; it will take any transistor schematic and produce a layout, one layer of abstraction lower.

Poirier describes a system called Excellerator which uses several heuristics for placing the transistors and a recursive A-Star maze router [Poi89]. It groups transistors together based on common gate inputs and diffusion abutments and then places these groups, using, among other algorithms, exhaustive search. The router, using the rip up and reroute method, routes the connections according to the priority determined from the schematic. This process is very expensive and this expense points to the complexity of the problem. Excellerator's results are promising: In [Poi89], the examples used were frequently smaller that the same cells laid out by human designers, and the advantage appears to mount as the transistor count of a cell increases.

Talib is an expert system which uses a knowledge base containing space-efficient layouts for the commonly appearing patterns of transistors [KMS84]. The system uses pattern recognition as it examines a schematic to identify patterns (for example, the complementary transistor pair forming a CMOS transmission gate) in the schematic's structure. Talib's knowledge base con-

tains space-efficient layouts for these patterns and it selects a layout appropriate to the current state of the layout process of the cell. The designer can set upper bounds on size for one or both axes. When the layout is constrained in one dimension, Talib's layouts are as good as or better than human experts most of the time. At worst, its layouts are 5% larger. When constrained in two dimensions, they are usually 15% worse, but can be up to 30% larger. Talib is intended for use on cells of up to 100 transistors.

Topologizer is an expert system which produces symbolic layouts by using separate placement and routing phases [KW85]. In the placement phase, Topologizer locates transistors on a virtual grid so that terminals which are to be connected together are adjacent. Topologizer's routing expert then routes the cell correctly, but not necessarily optimally. It then iteratively improves the routing. Topologizer usually produces cells which are marginally better or worse than those produced by human designers, although one example in [KW85] was 82% worse. Topologizer's knowledge base is oriented towards logic gates; its designers anticipate that supplementing it with rules about transmission gates would improve its performance on cells employing these.

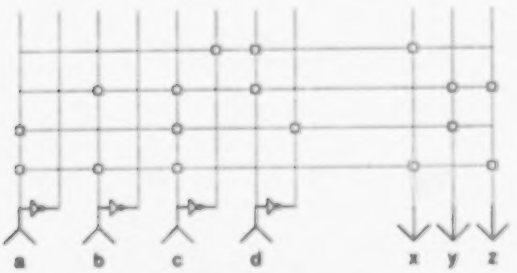
Play is another expert system [LM87]. Like Talib, it uses pattern matching, but employs a small set of primitives rather than a large library of common patterns. It is restricted to CMOS laid out in two rows of transistors and therefore lacks the generality of Talib.

### 2.1.3 Programmable Logic Array Generators

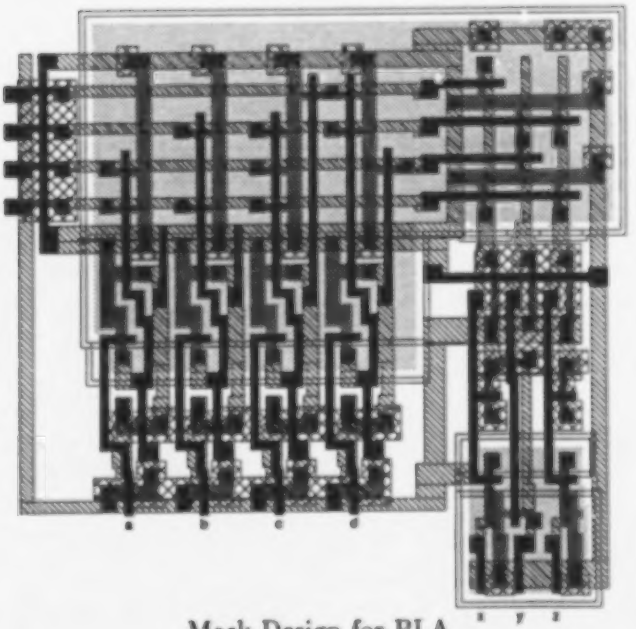
Programmable logic arrays (PLAs) are arrays of transistors which implement sum of products logic functions. PLAs are popular because they provide a convenient, structured method of realizing these combinational logic functions. The standard PLA consists of two planes of transistors — the "and" plane and the "or" plane. Input signals enter the "and" plane via its columns, where the products are formed. These signals are then sent to the "or" plane via the rows of the PLA. In the "or" plane, the sums are formed, and these exit the PLA via the "or" plane's columns. Three representations of a simple PLA design are shown in Figure 2.2. Three equations, with overlapping minterms, describe the function of this PLA. In the symbolic graphical form, each square shows the location of a transistor which connects either an input to the "or" plane, or an output to the "and" plane. The correspondence between the equations and the PLA can be readily seen. Tilers are frequently used to assemble PLAs because of their semi-regular, array structure. Tiles

$$x = abc + \bar{c}d$$
$$y = ac\bar{d} + bcd$$
$$z = abc + bcd$$

Boolean equations for a PLA




Symbolic PLA Design




Mask Design for PLA

Figure 2.2: Example of a PLA Design

are constructed both for positions in the planes which have and do not have transistors, as well as for input and output drivers and any other required structures. The tiler assembles the PLA from a structured specification, such as a character matrix which uses different characters to represent different tiles. The third part of Figure 2.2 shows a dynamic CMOS mask design for the PLA example, constructed with a tiler. A legend for the layers of the mask design can be found in Appendix A.

Pleasure is an example of a tool which takes a personality matrix (truth table) and creates a multiple folded PLA in symbolic form [MSV83]. It folds the PLA automatically, using two heuristics, or allows the designer to guide the process interactively. The symbolic form which Pleasure produces is another personality matrix which is read by a program such as Panda, which in turn constructs the mask for the PLA [MN84]. Panda is implemented with the tiler Tpack, which will be discussed below in Section 2.1.6 [MO83].

### 2.1.4 Weinberger Array Generators

Weinberger arrays are arrays of transistors arranged to form nor gates in NMOS technology [Wei67]. Metal is run vertically and diffusion is run horizontally. Transistors are located between the metal wires. Originally, these arrays were implemented with metal gate MOS; now, polysilicon gate technology is used. Thus, the polysilicon gates are connected to the vertical metal lines with contacts. Unlike PLAs, Weinberger arrays can implement combinatorial logic as multi-level nor networks, and generally require less area than the same function implemented as a PLA. Figure 2.3 shows the structure of a Weinberger array implementing exclusive-or.

Even though Weinberger arrays were first discussed in 1967 [Wei67], they continue to attract research interest. SWAMI is a system that takes in combinational logic or state machine descriptions, and produces Weinberger array layouts which implement the desired function [RH85]. Wong and Liu developed a simulated annealing based algorithm for generalized array folding which can be applied to Wienberger arrays, as well as to PLAs and gate matrix layouts [WL87]. Devadas and Newton developed a similar system [DN87]. Recently, a system was described which produces a row-folded Weinberger array from a register transfer language (RTL) description [SAK90]. As well, an algorithm for column-folding has also been published [SAR90]. These works attempt to find optimally dense Weinberger arrays.

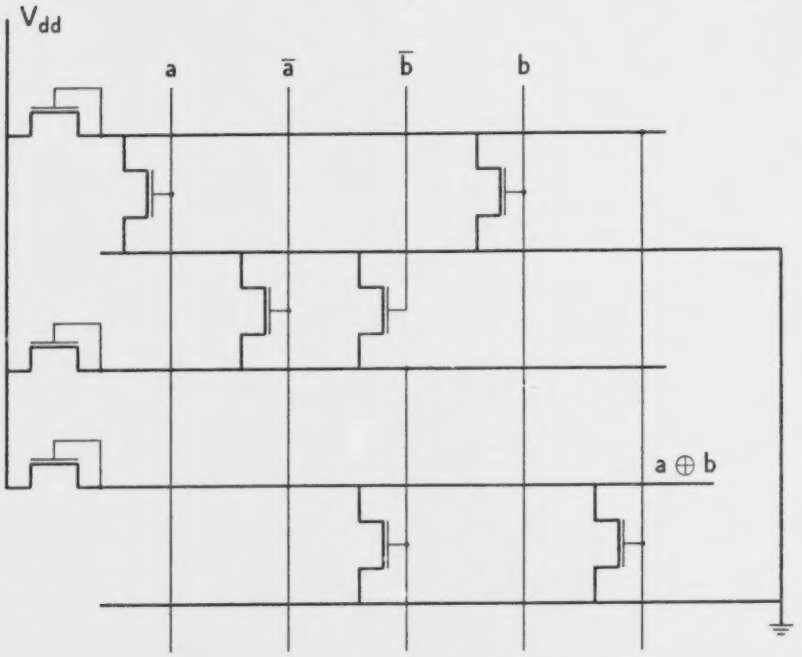



Figure 2.3: Weinberger Array implementing exclusive-or in NMOS

### 2.1.5 Gate Matrix Layout Tools

The gate matrix layout style is another semi-regular array style [LL80]. Polysilicon wires are run vertically, connecting transistors with a common gate input. Diffusion runs horizontally, where it is needed, as well as vertically, when short runners are required to connect with an adjacent row. Metal is run both horizontally and vertically. An example, in which a half-adder is implemented, can be seen in Figure 2.4. Essentially, the rules governing gate matrix are more relaxed than those for Weinberger arrays, since gate matrix is not restricted to multi-level nor structures. Finally, gate matrix works well with CMOS technology. Because of gate matrix's generality of circuit structure, and its regularity of layout, this style is highly favoured for algorithmic approaches to leaf-cell synthesis. Gate matrix layout is also used for layout of large, monolithic blocks of transistors [IO89]. This approach, infeasible for hand layout, is used in a number of integrated design and layout tools, and has been chosen because it performs well.

Both Wong and Liu's, and Devadas and Newton's, work on generalized

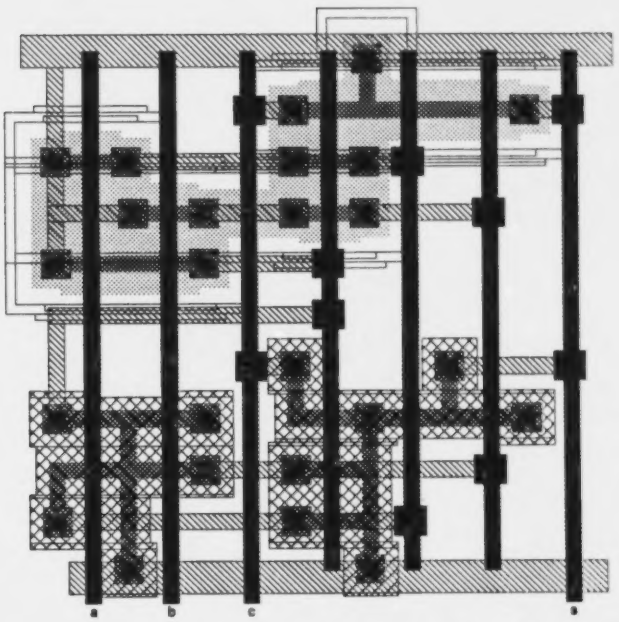



Figure 2.4: Example of a Gate Matrix mask design (after [LL80])

array folding (discussed in section 2.1.4 above) is applicable to gate matrix layout [WL87][DN87]. Huang and Wing described a set of algorithms tailored to gate matrix layout. These algorithms heuristically assign the gates to columns and then the nets to rows [HW89a]. Huang and Wing also developed an algorithm for partitioning a gate matrix into several smaller matrices, both reducing area from 11% to 41% and improving its aspect ratio [HW89b]. Thus, gate matrix is a practical and important design style for *Amoeba* to process.

### 2.1.6 Tilers and Iterated Structure Generators

A *tiler* is a program that assembles a module by combining previously designed sections (tiles) of layout. Traditionally, tilers have operated by abutting rectangular tiles together; hence their name. The tiles make connections to adjoining tiles by abutting their wires.

An *iterated structure* is implemented by combining semi-regular elements to form a one or two dimensional array. Unlike the structures referred to

here as semi-regular arrays, the elements of iterated structures tend to be larger, functional units as compared to semi-regular array elements. An example is an arithmetic logic unit for a micro-processor, which may consist of a series of bit slices which build up an adder, a multiplier and several registers. *Interated structure generators* implement iterated structures when given the parameters of the required module. The generators are frequently implemented as a control program built atop a tiler.

Tilers are also useful to higher level programs such as semi-regular array generators. They allow the array generator to output a simple, technology-independent description of an array. The tiler isolates the generator from the design rules of the target technology, thus simplifying generator design.

Some tilers accommodate different sized tiles by stretching the shorter tile until it is as big as the longer. These tilers require that the tiles be marked with lines to indicate where the additional space may be placed.

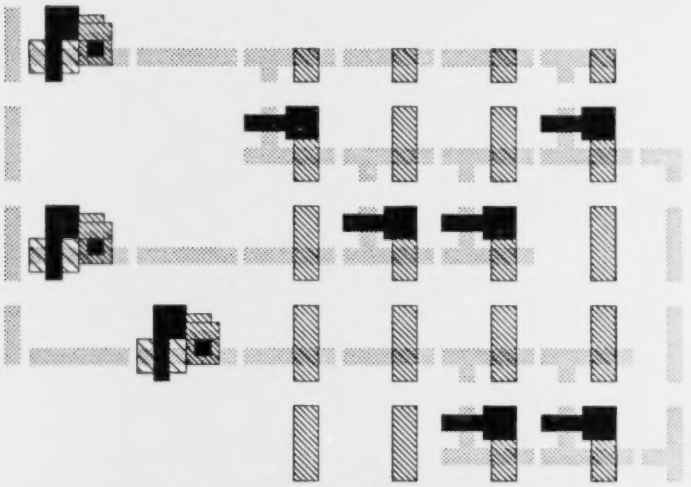
Tpack is a collection of subroutines for writing tilers [MO83]. It assumes that tiles are aligned by abutting the tiles at their boundaries and that connections are made by abutting the wires between these adjacent tiles. Tiles can also be overlaid on each other to form composite tiles.

Quilt is a tiler written using Tpack. It takes a simple personality matrix for the module to be assembled, and the tiles to be used in the matrix, and assembles them. Figure 2.5 contains a Weinberger array which was laid out by quilt. This is the same array shown in Figure 2.3. The first part of the figure shows the personality matrix used to generate the array. Each character is the name of a tile and the position of the tile in its module corresponds to the position of the character in its personality matrix. In the second part of the figure, the tiles, prior to abutment, can be seen. The final part of the figure shows the completed Weinberger array.

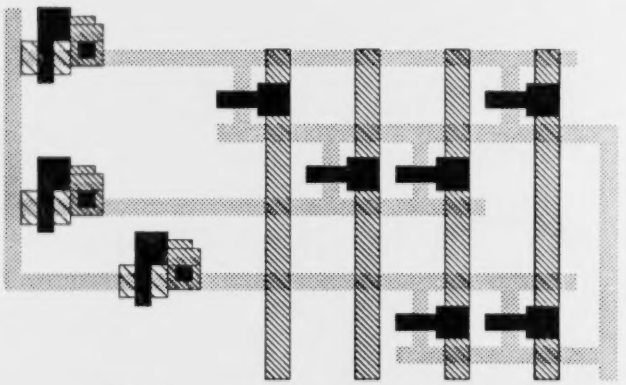
Tpla is a PLA generator written with Tpack. It can generate PLAs in either NMOS or CMOS processes. Tpla does nothing different for the different technologies — all the process dependencies are contained in the tiles themselves. The mask in Figure 2.2 was generated using Tpla, and Figure 2.6 shows the input file used to indicate to Tpla the format of the PLA. The first three statements specify the number of inputs, outputs and products in the array, respectively. The personality matrix follows, taking four rows in this example. The final statement signifies the end of the input. Frequently, PLAs will alternate two different tile types for odd and even rows and columns. For example, this can allow the PLA to use a single metal line to supply power or ground connections to two adjacent rows or columns, thus

```
acdfggfn
aeehiiho
acdjhhkp
bdcrrssp
leemmhhq
```

Personality matrix used as input for *quilt*



Tiles used by *quilt* to implement a Weinberger array



Weinberger array produced by *quilt*

Figure 2.5: Using *quilt* to build a Weinberger array[1]

```
.i 4
.o 3
.p 4
--01 1--
-111 -11
1-10 -1-
111- 1-1
.e
```

Figure 2.6: Input used by Tpla to generate the PLA in Figure 2.2

saving space. By defining the basic tiles as alternate pairs, Tpla, can support this design style. Even though only three symbols are used in the personality matrix (0, 1, -), Tpla correctly chooses tiles from a much larger set.

Bamji, Hauck and Allen developed a system called the Regular Structure Generator (RGS) [BHA85]. This system differs from Tpack because it allows hierarchical tiling, will rotate and mirror cells, and uses the design-by-example idea to specify the separation and orientation of two cells which may be located in adjacent positions. The design-by-example concept is further exploited by RGS's hierarchical tiling feature; superordinate cells inherit their separation and orientation characteristics from their constituent subcells. Inheritance simplifies the task of designing the tile set, while hierarchy makes this tool well suited for realizing iterated structures.

Law and Mosby describe a program for generating tilers in [LM85]. The programmer supplies a cell library and a floorplan of the array called the array structure template. The template is broken up into blocks, and each block has a list of properties associated with it, including how to map the tiler input to the cells in the library, what transformations to apply to the cells depending on their position in the block (such as mirroring or rotation) and an optional procedure for the tiler to execute to modify the block's layout.

Mocha Chip was designed as a graphical tool for designing iterated structure and semi-regular array generators [May86]. The designer uses Mocha Chip to make an assembly diagram which describes the structure of the module. The assembly diagram is comprised of an array of subcells, and a (usually) small amount of lisp code. Each subcell is itself an assembly dia-

[1]This array is implemented in NMOS. Thus, these patterns differ from the legend in Appendix A in two ways: here, the light screen is diffusion, and the coarse striped regions are the implants for depletion mode transistors.

gram or a piece of layout and the diagram shows the relative positions and orientations of the cells. Two special types of control cells are also defined; *array cells* allow a cell to be replicated in two dimensions and support the cell alternation described as part of Tpla, while *case cells* allow a cell to be chosen from a list of cells, depending on its context. In [May86], a PLA generator that is functionally equivalent to Tpla is constructed using Mocha Chip.

## 2.2 Amoeba

*Amoeba* has been developed to test several ideas:

- that a single tiler can be of use to all types of semi-regular array generators and iterated structure generators.
- that adding the concepts of wiring and compaction to a tiler will improve its flexibility.
- that allowing composite tiles to be specified by the user will ease the task of specifying the resulting array.

*Amoeba* is intended to be of use to all semi-regular array generators and iterated structure generators. It is designed to work with tiles as small as transistors and individual wires, and as large as functional components of an ALU. It should be able to assemble PLAs, Weinberger arrays and gate matrix arrays efficiently. It should also be able to lay out and compact a macro-cell comprised of large subcells.

A tiler can be used as the basis of an array or structure generator because of the regularity of the resulting module. Instead of having to specify the absolute positions of the elements of the array or structure, only relative positions need be given. The tiler can then generate the absolute positions and build the module. The same tiler may be used for more than one generator, with the obvious advantages of this efficiency accruing to the generator designer.

Separation of the functions of a tiler from those of an array or structure generator makes the design of the generator easier. The generator becomes independent of the design rules of the target process, because these are now handled by the tiler. The switch to a new process with smaller feature sizes, or to an entirely new process (such as from NMOS to CMOS) can be

accomplished with few or no changes to the generator. The designer of the generator can ignore all the mask details and concentrate on the topology of placement. The mask details are specified graphically to the tiler; the generator is specified as the designer sees fit.

In previous tilers, a row would have to be large enough to accommodate the largest feature which might be included in that row. Alternatively, the array generator, after ensuring that a large feature would not be placed in a row, would have to select a separate set of smaller tiles to reduce the size of the row. *Amoeba* tests the idea of using a single set of tiles and then compacting unused space out of a row. This flexibility is obtained through explicitly specifying the routing between tiles, so that the routing lengths can be adjusted.

A tiler that allows basic tiles to be overlaid to form composite tiles can simplify the design of array generators. For example, a gate matrix tool may specify a basic transistor tile wherever needed. If signals are to be routed from the drain or source on a layer other than diffusion, contact tiles can be overlaid on the source or drain, rather than substituting a separate transistor tile with contacts. In CMOS, eight tiles would be required if overlaid tiles are prohibited: in both $n$ and $p$ diffusion, a transistor with no contacts, two transistors with a contact on either source or drain, or a transistor with contacts on both terminals. If overlaid tiles are permitted, only four tiles are required: an $n$ and a $p$ transistor, and a contact for each side of the transistor.

## 2.3 Summary

Automated layout is the use of computer-based design tools to create mask designs from higher-level specifications. One method of achieving this is to divide a chip into functional sub-units called macro-cells. The macro-cells can be designed through the use of module generators, and then placed in the chip design and wired together by a global router. Several different types of module generators exist. Some are based on procedural generation of a function through the use of layout-languages. Others will take a structural specification like a transistor schematic and produce the mask design. The type of module generator that *Amoeba* is targetted for produces semi-regular arrays. Semi-regular array generators often leave the task of layout to a program called a tiler. These programs assemble the array by abutting cells

together.

*Amoeba* is a tiler, but it differs from the other in several ways. It is intended to function well with all types of semi-regular array generators. It compacts the array, saving area on the chip and it allows tiles to be overlaid to form composites, which simplifies the design of the tiles.

# Chapter 3

# Overview of Amoeba

*Amoeba* has been developed to explore ways of making tilers more effective by loosening some of the constraints tilers have placed on design in the past, and by adding the idea of compaction to tilers.

This chapter discusses *Amoeba's* operation from the perspective of a VLSI designer. First, an overview of *Amoeba* outlines the manner of specifying the array, and various options for compaction and routing. Then, the inputs and output of *Amoeba* are described.

## 3.1 Overview of *Amoeba*

*Amoeba* is a tiler, a program which assembles a module by placing predesigned pieces of mask layout into the rows and columns of a two dimensional *array*. These pieces of layout are called *cells*. *Amoeba* is supplied with a list of cells, and the locations at which they are to be placed. An array location is called a *site*. When one or more cells are assigned to a site (or sites, if the cells are too large for a single site), they form an *element*. Thus, *Amoeba* assembles an array of elements.

*Amoeba* allows multiple cells to be assigned to a single element. In other words, an element may contain a composite cell which is constructed from several simple cells. This feature of *Amoeba* can be used to simplify the library of cells needed to support a particular application of *Amoeba*. For example, a gate-matrix tool specifies a basic transistor tile wherever needed. If signals are to be routed from the drain or source on a layer other than diffusion, contact tiles can be overlaid, rather than substituting one of four different transistors.

Elements can span more than one position in the array. If an element is much larger than those surrounding it, it is advantageous to let it occupy more than one space. For example, if the array designer wishes to add a pair of latches to the inputs of an adder, he can save space by assigning two columns to the adder, and placing the latches above the adder, side by side. The latches will normally be smaller than the adder, and this layout makes efficient use of chip space.

Previous tilers have connected tiles together by abutment. When the tiles touch each other, the wires also touch, completing the inter-element connections. However, *Amoeba* uses explicit wires. A wire specifies its starting and ending terminals, and *Amoeba* connects these together. A *terminal* is a location on a cell to start or end a wire on. Each cell may have one or more terminals, all of which are named.

This explicit wiring gives the array designer more flexibility because fewer cells may need to be designed. Connections can be made to elements several array positions away, and the wires may even run overtop other elements.

Unlike other tilers, *Amoeba* compacts the array. *Compaction* is the process of squeezing unused space from a layout. *Amoeba* performs this as a pair of one dimensional compactions. First, the array is compacted along one axis, then the other. The designer specifies the order in which this is done. Space within the elements is not compacted; rather, *Amoeba* is restricted to sliding the elements as close together as the process technology will allow.

When the elements are squeezed together, it may be possible for one to slide so far that it appears to have changed columns or rows. *Amoeba* offers two modes of compaction — in one, this is not allowed, and in the other, it is. The term *rigid compaction* is used to describe the case where movement between rows and columns is not permitted. *Unconstrained compaction* describes the alternative. Rigid compaction is achieved by adding guidelines between the rows and columns. An element's bounding box may touch a guideline, but it must not cross it.

As *Amoeba* compacts the array along its first axis, elements may slide anywhere along the axis of compaction as long as they don't increase the size of the array, or interfere with other elements. However, this can create problems when *Amoeba* compacts along the second axis, particularly for the more regular structures, such as PLAs. In this case, rigid compaction is called for. The guidelines used in the compaction process ensure that the array retains its "row and column" appearance. If the elements are of varied sizes, however, allowing the elements to slide out of their strict array position can

save layout area. In this case, unconstrained compaction will better achieve the designer's goal.

When *Amoeba* compacts an array, the designer has the option of allowing or disallowing the wires to be bent. Bent wires allow greater freedom in compaction; however, the bends, called *doglegs*, take up space themselves. *Dogleg routing* is used to describe the case when the wires are bent; *straight routing* describes the other case.

Straight routing is recommended for more regular structures, such as PLAs, as was rigid compaction. The array will retain more of its regularity when straight routing is used. However, when working with an iterated structure, dogleg routing may improve the density of the layout by allowing *Amoeba* more freedom to compact the elements together.

*Amoeba* uses three types as input. These are the *array structure file*, the *design rule file* and the files from the *module directory*. The array structure file describes the structure of the array *Amoeba* is to build. It identifies the cells and their locations, as well as the connections between them. The design rule file describes the process technology to be used to compose the array, and includes the minimum separation between objects not connected together, and the width of the interconnecting wires. The module directory is a Unix[1] directory containing the cells from which the array will be constructed.

*Amoeba* writes out a file that describes the array. It locates the cells and routes the connecting wires between them. The output is formatted for use by *Algae*, a mask generator which converts cell-based descriptions to fully expanded mask descriptions.

## 3.2 Program Inputs and Outputs

This section will describe the input and output formats of *Amoeba* in detail. The array structure file, the design rule file, the module directory and the output describing the compacted array are all included.

### 3.2.1 Array Structure File

The array structure file is the primary input to *Amoeba*. It specifies the general layout of the array, and the connections between its elements. Figure 3.1 contains an example of an array structure file and is helpful when reading

[1]Unix is a trademark of AT&T Bell Laboratories.

the description of the file below while Figure 3.2 formally describes its syntax. The notation used in Figure 3.2 is explained in Appendix B.

A element is placed by the *cell* statement. This statement describes the location, size and composition of an element. It also describes any wires which originate from this location. A cell statement begins with a %C. The column and row numbers where the cell is to be placed follow. Their numbers start at zero and column numbers increase to the right, while row numbers increase upward. If the element is larger than a single column and row, its size, measured in columns and rows is also included. Both the location and size of the cells are determined by the program's user. The cells to be placed in this element are next, followed by a list of wires. Each wire is described by the terminal it starts on (in this element), an indicator of the direction it runs, and the name of the terminal on which it ends. The direction indicator is either an exclamation mark (!) or a greater-than sign (>). After the array structure file has been read, *Amoeba* will search for a terminal with the same name as the end terminal of this wire. It will search in the indicated direction: up, for an exclamation mark, or to the right, for a greater-than sign, in the same row or column as the originating element. It will select the first terminal that it finds with the specified terminal name and connect the two terminals with a wire.

If the element is larger than a single row and column, the wires specified in the cell statement will run in the first row or column occupied by the element. This may affect which element *Amoeba* selects to terminate the wire. If the array designer wishes to place the wire in a different row or column, the *oversize* statement is used. This statement consists of %O followed by the row and column in which to originate these wires, the list of cells, and the wires to run in this row and column. This format closely follows the *cell* statement.

Comments may also be included in the file. An octothorpe (#), and all text following it to the end of the line, are ignored by *Amoeba*.

Figure 3.1 contains a short example of an array structure file. The first three lines are comments, which are ignored by *Amoeba*. They are followed by three cell statements. The first statement places a cell called register in the element at the intersection of the first column and the first row. Several signals are run out to a cell to the right of this one, and one signal runs to a cell above this one. The third cell statement places a cell into two array locations, corresponding to row two and columns one and two. It is followed by an oversize statement, which specifies that a wire is to run up from that cell, in column two. The next statement shows two cells being stacked on

```
#
#   Example for Chapter 3
#
%C 1 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 2 1 register; out!y
%C 1 2 2 1 comp;
%O 2 2 comp; yprime!y
%C 1 3 2 1 fadder1 fadder2;
```

Figure 3.1: Example of an Array Structure File

each other. They occupy two columns, as does the cell located below them in the array.

### 3.2.2 Module Directory

The module directory contains the set of cells that can be placed in an array. It is organized as a Unix directory and each cell is its own file in that directory.

The files are in Magic format [SMHO86]. Magic is a mask editor developed at the University of California at Berkeley. A designer draws out the cell with Magic, and saves it. If cells are to be stacked together, *Amoeba* will ensure that the cells' origins are aligned.

Terminals are defined with the Magic *label* command. A rectangular region is identified with Magic's cursor, and a label is attached to this region. *Amoeba* interprets all such labels as terminals and it will attach wires to the centres of labels. Magic will usually associate a label with the process layer below the region selected with the cursor. This is not always a correct interpretation. For example, if the designer wishes to create a terminal on the n-diffusion layer, attached to a n-diffusion-to-metal contact, Magic will associate the label with the contact layer, rather than with the diffusion. To avoid this problem, the designer can add a prefix to the terminal name. If a terminal is to be called `in1`, the designer can enter `nd_in1`. If the prefix `nd` is defined in the design rule file as indicating the n-diffusion layer, *Amoeba* will ignore Magic's association and use its own.

*arrayStructure* → *arrayStructure elementDescriptor* <newline>
| *arrayStructure* <newline>

*elementDescriptor* → %C *cellElement*
| %O *oversizeElement*

*cellElement* → *columnPosition rowPosition columnSize rowSize cellList* ; *connectionList*
| *columnPosition rowPosition columnSize rowSize cellList*
| *columnPosition rowPosition cellList* ; *connectionList*
| *columnPosition rowPosition cellList*

*oversizeElement* → *columnPosition rowPosition cellList* ; *connectionList*

*columnPosition* → **integer**

*rowPosition* → **integer**

*columnSize* → **integer**

*rowSize* → **integer**

*cellList* → *cellName*
| *cellList cellName*

*cellName* → **identifier**

*connectionList* → *connectionSpecifier*
| *connectionList connectionSpecifier*

*connectionSpecifier* → *terminalName direction terminalName*

*terminalName* → **identifier**

**identifier** → [A-Za-z_][A-Za-z0-9]*

**integer** → [0-9]+

**direction** → >
| !

Figure 3.2: Array Structure File Syntax

```
c Design Rule File for CMOS3d -- @(#)DesignRules 3.1 11/2/90 15:30:05
w 5
p m1 metal1
p m2 metal2
p p polysilicon
p nd ndiffusion
p pd pdiffusion
r metal1 metal1 5
r metal1 polycontact 5
r metal1 ndiffcontact 5
r metal1 pdiffcontact 5

:

r ntransistor pdiffusion 21
r ntransistor ptransistor 21
r ntransistor pdiffcontact 21
r ndiffcontact ndiffcontact 7
r ndiffcontact pdiffusion 17
r ndiffcontact ptransistor 17
r ndiffcontact pdiffcontact 17
```

Figure 3.3: Portions of a Design Rule File

### 3.2.3 Design Rule File

The design rule file describes those aspects of the target process technology that *Amoeba* needs to know. This includes the minimum separation between objects, the associations between terminal prefixes and the layers, and the width of the wires to be used. Each statement in the design rule file begins with a single character, and ends with a newline. Figure 3.3 contains portions of a design rule file, and Figure 3.4 contains a detailed syntax diagram for the design rule file.

Comments can be written in the design rule file by preceding them with a `c`. In Figure 3.3, the first line is a comment.

A width statement specifies the width of the wires to be used when connecting the elements together. It is comprised of a `w` followed by the width expressed in microns, and can be seen as the second line in Figure 3.3.

In Section 3.2.2 above, Magic's *label* command and the requirement of assigning a layer to a terminal was discussed. One method of doing this is

| | | |
|---|---|---|
| *design-rule-file* | → | *statement design-rule-file* |
| | \| | *statement* |
| *statement* | → | *rule-statement* |
| | \| | *prefix-statement* |
| | \| | *wire-statement* |
| | \| | *comment-statement* |
| *rule-statement* | → | **r layer layer separation** <newline> |
| *prefix-statement* | → | **p prefix layer**<newline> |
| *wire-statement* | → | **w width** <newline> |
| *comment-statement* | → | **c comments** <newline> |
| **layer** | → | <any character except <space>>+ |
| **separation** | → | [0–9]+ |
| **prefix** | → | <any character except <space>>+ |
| **width** | → | [0–9]+ |
| **comments** | → | <all characters except <newline>>+ |

Figure 3.4: Design Rule File Syntax

to use a label of the form `nd_in1`, where `in1` is the terminal name, and `nd` is an abbreviation of the layer name. A prefix statement is used to associate a prefix (`nd`) with a layer (n-diffusion). There are five prefix statements in Figure 3.3, all of which begin with `p`.

A rule statement specifies the minimum allowable distance between objects on two particular layers. These rules are used to calculate the minimum separation between two elements. A bounding box is calculated for all geometries on each layer for each element. A bounding box is the smallest rectangle which will enclose all the features under consideration. By examining the adjacent edges of all pairs of bounding boxes (one from each element), *Amoeba* finds the minimum separation of the elements, such that all the rules are complied with. This separation is used to place the elements. A rule statement includes an `r`, (to identify it as such), the names of the two

layers for which the rule applies, and the minimum separation to be enforced between them. Figure 3.3 contains several of these statements.

### 3.2.4 Algae Output

*Amoeba* writes its output in Algae format. Algae is a mask generator developed at the University of Alberta [Mow87]. It takes the names and locations of cells and the path and size of wires and produces a description of the layout in Magic format [SMHO86]. A description of the subset of Algae used by *Amoeba* is found in Figure 3.5, and some Algae produced by *Amoeba* is shown in Figure 3.6.

A description of a mask in Algae format, as produced by *Amoeba*, comprises five types of statements. They are the cell descriptor, routing region, segment, comment and end statements.

A cell descriptor statement names the cell to be placed, locates its origin, describes any rotation or mirroring to be applied, and optionally, the size of the cell. It starts with a `C` and is followed by the name of the cell. The $x$ and $y$ coordinates of the cell's origin follow. Then, the rotation of the cell is included, as two numbers which describe the rotation angle in terms of its abscissa and ordinate. This is followed by a single character which selects the cell mirroring: either none (`N`), mirrored about the $x$ axis (`X`), or mirrored about the $y$ axis (`Y`). The statement may optionally have the size of the cell appended.

A routing region statement must immediately precede any sequence of segment statements in the file. It establishes a relative origin for the subsequent segment statements. A routing region statement starts with a `R` which is followed by a label to identify the region. The origin of the region ends the statement. *Amoeba* always describes wire routes relative to the array origin, rather than the element origin, as this information is directly produced by *Amoeba*. Therefore, all of its routing regions take the form "`R A 0 0`."

A segment statement describes a straight segment of a wire. It names the layer on which the wire runs, its width, and its start and end points. Multiple segments must be used to describe a wire with jogs. This statement starts with an `S`, followed by the layer and width of the segment. The start and end points follow.

A comment statement is ignored by Algae. It is used to annotate the file. Comment statements are delimited by a pair of parentheses; anything between them is discarded.

| | | |
|---|---|---|
| *algae-output* | → | *statement-list* `E` |
| *statement-list* | → | *statement statement-list* |
| | \| | *statement* |
| *statement* | → | *cell-descriptor* |
| | \| | *routing-region-descriptor* |
| | \| | **comment** |
| *cell-descriptor* | → | `C` *cell-name origin rotation mirroring extent* |
| | \| | `C` *cell-name origin rotation mirroring* |
| *routing-region-descriptor* | → | *routing-region segment-list* |
| *routing-region* | → | `R` *label origin* |
| *segment-list* | → | *segment segment-list* |
| | \| | *segment* |
| *segment* | → | `S` *layer width start-coordinate end-coordinate* |
| **comment** | → | `(` `[^()]+` `)` |
| *cell-name, label, layer* | → | **string** |
| *origin* | → | **integer integer** |
| *rotation* | → | **integer integer** |
| *mirroring* | → | `[MXY]` |
| *extent* | → | **integer integer** |
| *width* | → | **integer** |
| *start-coordinate* | → | **integer integer** |
| *end-coordinate* | → | **integer integer** |
| **string** | → | [^<space><tab><newline>]+ |
| **integer** | → | `[0-9]+` |

Figure 3.5: Subset of Algae Used by *Amoeba*

```
(Algae written by ../src/am on Wed Jul 17 15:27:28 1991)
(element at [1,1])
C m1 0 0 0 0 N 500 500
(element at [1,5])
C m1 0 5200 0 0 N 500 500
(element at [2,1])
C m1 1000 0 0 0 N 500 500
(element at [2,2])
C nfet 800 700 0 0 N 1300 1500
C nfetlc 800 700 0 0 N 900 900
(element at [2,3])
C m1 3200 2800 0 0 N 500 500
(element at [2,4])
C pfet 800 3500 0 0 N 1300 1500
C pfetlc 800 3500 0 0 N 900 900
C pfetrc 800 3500 0 0 N 900 900
(element at [2,5])
C m1 1000 5200 0 0 N 500 500

⋮

(wires originating at [3,4])
R A 0 0
S pdiffusion 500 2800 4250 4800 4250
R A 0 0
S polysilicon 500 5250 1950 5250 3750
(wires originating at [3,5])
R A 0 0
S metal1 500 1250 5450 6350 5450
R A 0 0
S metal1 500 6350 4250 6350 5450
(wires originating at [4,1])
R A 0 0
S metal1 500 1250 250 7350 250
(wires originating at [4,5])
R A 0 0
S metal1 500 6350 5450 7350 5450
E
```

Figure 3.6: Example of Algae Produced by *Amoeba*

Algae input is terminated with the end statement. It consists, simply, of a E.

# Chapter 4

# Constraint Generation

## 4.1 Formulating Layout as Linear Programming

*Linear Programming* is a powerful technique for solving optimization problems which consist of linear constraints and a linear cost function. As framed by *Amoeba*, layout can be cast as a linear program (LP). To do so, layout constraints must be expressed in a form suitable for LP.

In this chapter, linear programming will be reviewed and *Amoeba's* layout constraints will be expressed as LP constraints. The method of calculating the minimum separation between two features will be explained, as well as the algorithms for determining which features might interfere, and must therefore have constraints generated between them.

### 4.1.1 Linear Programming in Standard Form

The *Standard Form* of an LP is

$$\begin{aligned} & \min c'x \\ Ax &= b \\ x &\geq 0 \end{aligned} \tag{4.1}$$

where $A$ is an $m \times n$ integer matrix, $c'$ is the transpose of a vector $c$ of size $n$, $b$ is a vector of size $m$, and $x \in R^n$ ($x_i$ will represent the $i$th scalar in $x$). The requirement that $A$ be an integer matrix is derived from results about the stability of linear programs. In practice, almost all matrices can be scaled to produce an integer matrix.

All constraints in an LP are linear, and both equality and inequality constraints can be accommodated. The method of representing equality constraints is obvious in the definition above; include an equation such as

$$a_{i1}x_1 + a_{i2}x_2 + \ldots = b_i$$

in

$$Ax = b$$

Inequalities can be represented by the introduction of a *slack variable* into the inequality. For example,

$$x_{ij} \geq b_i$$

can be changed into

$$\begin{aligned} x_{ij} - s_{ij} &= b_i \\ s_{ij} &\geq 0 \end{aligned}$$

Thus inequality constraints can be handled as easily as equality constraints; the restriction $s_{ij} \geq 0$ is incorporated into $x_{ij} \geq 0$ by treating $s_{ij}$ as $x_{(i+k)j}$. In other words, once the slack variable has been generated, no changes to the algorithm used to solve the LP are required.

The expression,

$$c'x,$$

in the standard form, is called the *cost function*. The objective of LP is to minimize this function, without violating any of the constraints. The cost function is simply a linear combination of the variables of the LP.

### 4.1.2 Geometrical Interpretation of Linear Programming

The constraints and cost function of an LP have an intuitive geometrical interpretation. Each equality constraint defines a hyperplane in the solution space. Each inequality constraint defines a half-space. The cost function behaves as a linear potential function in this space. Feasible solutions, those which do not violate any constraints, are contained in the intersection of all the hyperplanes and half-spaces of the constraints. This intersection is a polytope.

The optimal solutions to the LP, called *optimal feasible solutions*, are those with minimal cost function values within the polytope. These are contained by a face of the polytope, which is either a vertex, or an edge or

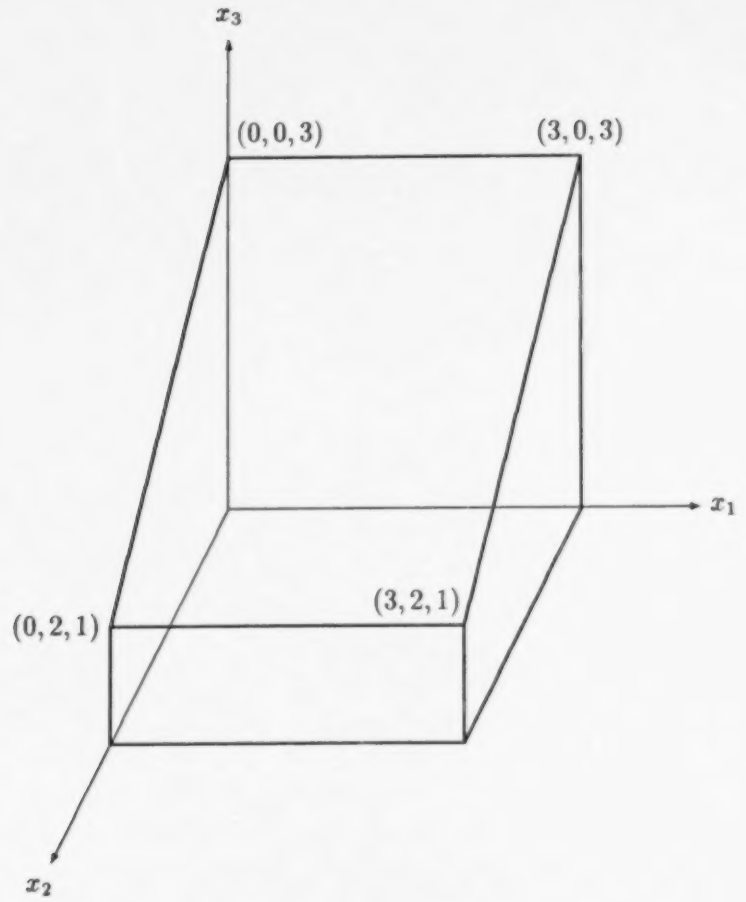



Figure 4.1: Example of a Polytope

a facet of the polytope. The simplex method solves an LP by traversing the exterior of the polytope, moving from vertex to vertex, each time reducing the value of the cost function.

In Figure 4.1, a polytope in 3-space, corresponding to a three constraint LP, is illustrated. The constraints which create this polytope are:

$$\begin{aligned} x_1 &\leq 3 \\ x_2 &\leq 2 \\ x_2 + x_3 &\leq 3 \\ x &\geq 0 \end{aligned}$$

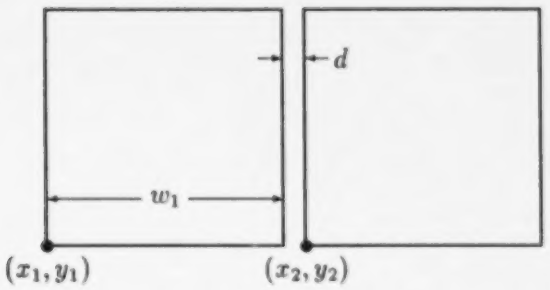



Figure 4.2: An Interference Constraint

The point at $(3,2,1)$ is a vertex on the polytope, and the line segment running from $(0,2,1)$ to $(3,2,1)$ is an edge. The rectangle with the four corners $(3,0,3)$, $(0,0,3)$, $(0,2,1)$ and $(3,2,1)$ is a facet. If the cost function

$$x_1 - x_2 + x_3$$

is added to this example, the optimal feasible solution to the resulting LP is found at the vertex $(3,0,3)$.

### 4.1.3 Tiling as Linear Programming

There are three problems to be solved when representing tiling as LP. Interference constraints, rigid wiring constraints and the cost function must all be represented in the LP.

*Interference constraints* enforce the minimum separation required between elements so that the layout can be fabricated. Figure 4.2 contains a simple example of this type of constraint. The positions of the elements are described by the position of their origins. Thus, $w_1$ is the distance from the origin to the far side of the first element and $d$ is the minimum separation that must be enforced between the bounding boxes of two elements. In this example, the position of the two the elements are constrained by

$$x_2 - x_1 \geq w_1 + d$$

With the addition of a slack variable, this constraint becomes

$$x_2 - x_1 - s = w_1 + d$$

Since $w_1 + d$ is constant, this constraint is in standard form.

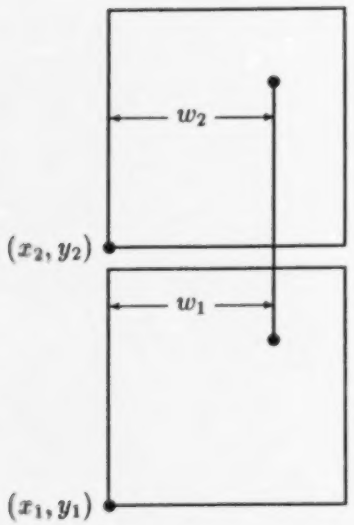



Figure 4.3: A Straight Wiring Constraint

*Rigid wiring constraints* are imposed when *Amoeba* is used in its rigid wiring mode. If wires must not be bent, then connected elements constrain each other's positions. Figure 4.3 shows two elements connected by a wire. In this situation, the two elements constrain each other with

$$x_1 - x_2 = w_2 - w_1$$

This is in standard form.

*Amoeba's* goal is to minimize the the length of each side of the array. This goal is easy to map onto a cost function suitable for LP. In Figure 4.4, a simple array is used as an example. The first step in generating a cost function for this array is to create a guideline outside all the elements. *Amoeba* must constrain this guideline so that it does not overlap any elements. Then, if the location of this guideline is $x_c$, the cost function is simply $x_c$.

### 4.1.4 Solving Linear Programs

By far the most popular algorithm for solving LPs is the simplex method. Despite its theoretical shortcomings, it performs well for small and medium sized problems and is easy to program. For these reasons, *Amoeba* uses the simplex method. In this section, the simplex method is reviewed, and several recent developments by Khachiyan and Karmarkar are discussed.

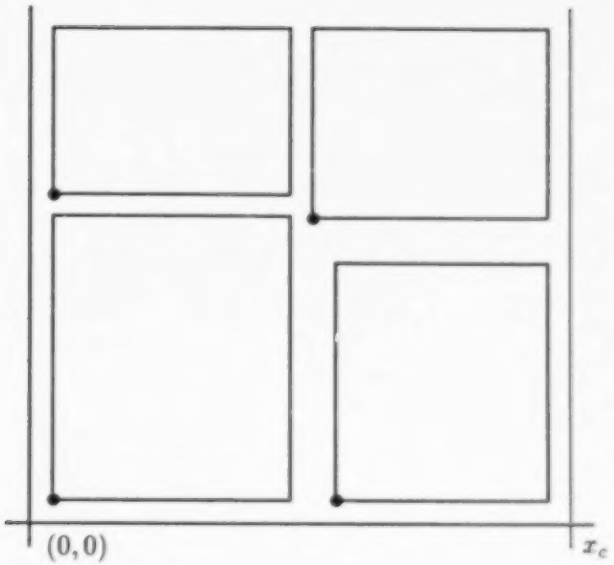



Figure 4.4: *Amoeba's* Cost Function

### The Simplex Method — The Data Structures

The constraints and cost function of an LP form a system of linear equations, suitable for representation as a matrix. The matrix format used to represent an LP is called a *tableau*, in which the cost function and each constraint occupy separate rows. Each variable occupies a column. The simplex engine written for *Amoeba* numbers its rows and columns starting from zero. Column zero is used for the constants of the constraints and row one for the cost function. The purpose of row zero will be discussed below.

As an example, recall the polytope illustrated in Figure 4.1. It represented these constraints:

$$\begin{aligned} x_1 &\leq 3 \\ x_2 &\leq 2 \\ x_2 + x_3 &\leq 3 \end{aligned}$$

When converted to standard form, this system becomes

$$x_1 - x_4 = 3$$

$$\begin{aligned} x_2 - x_5 &= 2 \\ x_2 + x_3 - x_6 &= 3 \end{aligned}$$

where $x_4$, $x_5$ and $x_6$ are slack variables. If the cost function is

$$x_1 + 2x_2 - 4x_3$$

then the resulting tableau would be

$$\begin{array}{rrrrrrr} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & -4 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & -1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & -1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & -1 \end{array}$$

This example reveals a characteristic of almost all tableaux: they contain many zeros. Zeros, of course, have the useful property that when they are multiplied by anything, they remain zero, and when any number is added to them, the number added simply replaces the zero. This can be exploited by a well programmed simplex routine to save memory space and shorten solution time.

A matrix whose terms are mostly zero is called a *sparse matrix*. Sparse matrices are not usually represented in computer programs as conventional arrays. Rather, only the non-zero elements are stored. In *Amoeba*, sparse matrices are implemented with two vectors, one called the row header, and the other the column header. The row or column number is used as an index to these vectors. Each header element points to a linked list of all the non-zero values (and their indices) in that row or column. If the list element corresponding to the desired matrix element cannot be found, it is zero. Matrix elements are updated by changing, deleting or adding list elements as appropriate.

The sparse matrix representation saves time as well as space. A simple example is the task of multiplying a row by a constant. Given a row with 100 columns, of which only four are non-zero (a common situation with *Amoeba*), the sparse matrix method requires only four multiplications. The conventional array representation would be programmed as 100 multiplications, or at least four multiplications and 100 comparisons.

### The Simplex Method — The Algorithm

This section presents a practical review of the simplex method. For a more complete and theoretical treatment of Simplex, see [PS82] or [Dan63].

In the standard form of an LP (Equation 4.1), $A$ is an $m \times n$ matrix. This review of the simplex method begins with the assumption that there are $m$ linearly independent columns in the tableau (and therefore in $A$). This assumption will be treated in more detail below. The $m$ columns are called a *basis* or *basic set*. The columns not included are called the *non-basic set*. If the variables not included in the basis are set to zero, Gaussian Elimination can be used to obtain the values for the variables included in the basis. This value of the vector $x$ is called a *basic solution*. If the basic solution does not violate any of the constraints, it is called a *basic feasible solution*.

An arbitrary basic feasible solution is selected as a starting point. The simplex method operates by exchanging columns between the basic and non-basic sets. The non-basic column with the most negative cost function coefficient is chosen to enter the basis. Then a row is selected so that simplex may pivot the tableau. This is done by taking each coefficient that is greater than zero in the selected column, and dividing it into the constant term of that row (i.e. column zero). The row with the smallest quotient is selected. New non-basic columns are chosen to enter the basis using this same procedure until all of the cost function coefficients are positive or zero.

Since *Amoeba* does not generate an LP with an obvious basic feasible solution, the first step is to find one. *Amoeba* appends a unit matrix to the end of the tableau. These new variables are called *artificial variables*. A new cost function is inserted into row zero which gives every artificial variable a cost of one, and all the others, zero. Simplex is then used to minimize these variables. If not all are zero at the end of this Simplex run, then there is no feasible solution to the original LP. If all are zero, however, the columns of non-basic artificial variables are deleted from the tableau. The basic artificial variables are constrained to be equal to zero so that their values will not change in the next phase of the simplex method. In this way, the tableau is left ready to rerun the simplex routine and solve the LP.

### Alternatives to Simplex

Simplex is by far the most popular method for solving LPs. It is surprising, therefore, to find that it is, in the worst case, exponential in time complexity [PS82][DB89]. However, it exhibits linear behaviour in the typical case

[DB89].

Simplex is an exterior method; that is, it finds the solution by traversing the exterior surface of the polytope. This, combined with its worst case behaviour, has motivated researchers to search for a worst case polynomial time, interior method for solving LPs.

In 1979, L. G. Khachian published a paper in which he described the *ellipsoid method* [Kha79]. This method operates by defining an ellipsoid which encloses the polytope representing the LP. In successive steps, the ellipsoid is bisected, and the half ellipsoid which encloses the solution to the LP is itself enclosed in a new ellipsoid smaller than the original one. These steps are repeated until a solution is found, or until a certain number of steps have been completed, from which it can be known that no feasible solution exists.

While the ellipsoid method is polynomial in the worst case, which is better than simplex's exponential worst case, its running times are disappointingly long. Many iterations must be completed, and each iteration is computationally expensive. When compared to simplex's typically linear behavior, the ellipsoid method is impractical.

Subsequent to Khachian's work, Karmarkar has developed another interior method [Kar84]. Using a projective transformation, his method succeeds in finding the optimal solution to the LP in a very small number of iterations, compared to those needed by the simplex method. But again, the computational expense of each iteration is extreme, at least for current implementations. None of the early implementations described in the literature were substantially faster than simplex, and all were much slower for certain problems. It has been reported that certain other implementations are, on average, faster than simplex. However, Karmarkar's method still provides fertile ground for more research [DB89].

The simplex method was selected as the algorithm to solve *Amoeba's* LPs because it performs competitively with other algorithms, and is easily implemented.

## 4.2 Constraint Generation

The core of the *Amoeba* research is the constraint generation problem. Input and output formats are largely syntactic issues; the simplex method is well understood. However, the selection of feature pairs to be used for constraint

generation and efficient generation of the constraints controls the efficiency, and therefore the usefulness, of *Amoeba*.

Constraint generation comprises three distinct tasks. The first is efficient generation of a constraint, to be placed in the tableau, given two features (elements or wires) which may interfere with each other. The second is generation of wiring constraints. The third is selection of only those features which may interfere with each other. For example, if three polysilicon wires are laid out side by side parallel to the $y$ axis, only two constraints are required to ensure that the design is free from design rule violations. One of these is a constraint between the left-most and the centre wires and the other is between the centre and the right-most wires. A constraint between the left-most and right-most wires is redundant and should not be generated because the simplex method will solve the LP in less time without it.

### 4.2.1 Interference Constraint Generation

There were three decisions which had to be made when designing the portion of *Amoeba* which generates a constraint in order to prevent two features from interfering with each other. The first was the representation of each member of the pair of features, and their bounding boxes. The second was the representation of the design rules which specify the minimum separations between two process layers. Finally, an algorithm was needed to examine these data structures and generate the constraint.

An array element may contain a number of cell instances. Each cell instance points to the description of its cell. The description is unique — if several instances of a particular cell exist in an array, all point to the same description. The description of a cell contains its bounding box list. The bounding box list contains the bounding box of features for each process layer. However, for efficiency, after the array description and the individual cell descriptions are read in, a bounding box list is made for each array element. This bounding box list is made by merging the bounding box lists of the constituent cells of the array element.

A bounding box is a simple structure, consisting of the coordinates of the lower left and upper right corners and the process layer for which the bounding box applies. A bounding box list contains a bounding box for the geometries on each process layer used by the element and a bounding box for all layers in the element. The bounding box for all layers of the the entire element or cell is represented by a bounding box with a null layer name.

```
InterferenceAlgorithm(B1, B2, R)
B1: list of bounding boxes for first element, sorted in order of the boxes'
  layers
B2: list of bounding boxes for second element, sorted in order of the boxes'
  layers
R: list of separation rules, consisting of two layers, a separation, and a
  pointer to the next rule, sorted in order of both layers
    {
    c = -∞;
    r = first(R);
    for each bounding box b1 from B1
        {
        while r ≠ ∅ and layer(b1) > layer1(r)
            r = next(r);
        if r ≠ ∅ and layer(b1) = layer1(r)
            for each bounding box b2 from B2
                while r ≠ ∅ and layer(b1) = layer1(r)
                    {
                    if r ≠ ∅ and layer(b2) > layer2(r)
                        r = next(r);
                    if r ≠ ∅ and layer(b1) = layer1(r) and layer(b2) =
                      layer2(r)
                        c = max(c, separation(r));
                    }
        }
    return c;
    }
```

Figure 4.5: Interference Constraint Generation Algorithm

The bounding box lists are kept in order of the layer names to speed the constraint generation algorithm.

A design rule is represented as the names of the two layers to be separated and the minimum separation to be enforced between them. A design rule list which describes all the rules for a given fabrication process is maintained. It is kept sorted in order of the first, and then the second layer name.

The pseudo-code for the algorithm which generates the constraint for a pair of elements is shown in Figure 4.5. All possible pairings of bounding boxes are generated by cycling through both bounding box lists. As each pairing is generated, the design rule list is checked for a rule. Because the pairings are generated in sorted order, the design rule list is not searched, but rather is advanced through sequentially. After completing the cycling, the largest minimum separation is used to generate a constraint. This algorithm is $O(n^2)$, where $n$ is the number of layers in the process.

### 4.2.2 Wiring Constraint Generation

Rigid wiring constraints are generated by iterating through the wires attaching the element to elements to the right, or above the current element. The constraint is calculated as shown above in Section 4.1.3, and inserted into the tableau.

When wires run from one element to another and the two elements are separated by one or more array sites, it is possible for the wire to interfere with features below or on its left. To avoid this, each wire must be treated like an element, and be checked against the features already processed. This is done as part of wiring constraint generation, but is handled in an identical fashion to interference constraint generation.

## 4.3 Feature Selection

In any array of meaningful size, the decision as to which constraints to include in the tableau is more important to the running time of *Amoeba* than how long it takes to generate an individual constraint. Most of the design effort that went into *Amoeba* was spent on the algorithms which select features for constraint generation.

In all the discussions that follow, only the columnwise case of and routing (compaction parallel to the $x$-axis) will be examined in order to remove

gratuitous differences in terminology and focus on the algorithmic differences arising from first and second pass processing of the array.

The first pass of *Amoeba* is different from subsequent passes because certain dimensions which could influence compaction have not been calculated yet. The separation between the array elements in the vertical direction is a primary example of this. To resolve this problem, the rows of the array are assumed to be separated by an infinite amount. In other words, the rows are sufficiently separated to guarantee that no interference can occur between elements in different rows.

In the second pass, the separations which were unknown in the first pass have been calculated, and a new problem must be addressed. There exists in the second pass the possibility of elements in different rows interfering with each other. Thus, when compacting the columns together, *Amoeba* must look for interference not only with cells in the same row, but also in neighbouring rows as well.

All eight permutations of compaction, wiring type and pass use the same basic algorithm, shown in Figure 4.6, to select features for constraint generation. Initially, the data structures needed by the constraint generation routines are set up and a guideline is created to prevent elements from overlapping the line $y = 0$. The simplex algorithm will ensure that the element origins will not overlap this line; however, geometry in an element is permitted to the left of the element's origin.

Having completed the initialization, the algorithm processes each column, from left to right, and each site in each column, from bottom to top (both are in ascending order of row or column number). Essentially, as each element is processed, the algorithm "looks back" at the features to the left, and generates constraints when those features could interfere with this element.

The constraints are generated by the two procedures GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$) and GenerateWiringConstraints($A, i, j, e_{m,n}, D$). The parameter $A$ is the entire array. This allows the procedures to "look around" the given element for other features. The current position in the array is described by $i$ and $j$ and the current element is $e_{m,n}$. The subscripts on $e$ will not be the same as $i, j$ if the element occupies more than one array site. If this is the case, it is necessary to know the lower left corner of the element. Two types of data structures will be explained below; these are the *feature table* and the *interval grove*. Either one may be used by SelectionAlgorithm(), depending on which permutation of compaction, wiring type and pass the algorithm is used for. $D$ refers to the structure appropriate to

```
SelectionAlgorithm(A)
A: the array to be processed by Amoeba;
    {
    initialize data structures D;
    create a vertical line v_0 and constrain its position to the origin;
    for each column c_i in A
        {
        if rigid compaction is being performed
            create a vertical line v_{i+1} and constrain it to remain on the
              right of v_i;
        for each site a_{i,j} in column c_i
            if site a_{i,j} is occupied by an array element e_{m,n}, which is k
              columns and l rows in size and its lower left corner occupies
              a_{m,n}
                {
                if i = m and j = n
                    {
                    if rigid compaction is being performed
                        generate a constraint to keep e_{m,n} to the right of
                          v_i;
                    GenerateInterferenceConstraints(A, i, j, e_{m,n}, D);
                    GenerateWiringConstraints(A, i, j, e_{m,n}, D);
                    }
                if rigid compaction is being performed and j = n and
                  i = m + k - 1
                    generate a constraint to keep e_{m,n} to the left of v_{i+1};
                }
        for each array site a_{i,j} in column c_i
            if site a_{i,j} is occupied by an array element e_{m,n}, which is k
              columns and l rows in size
                {
                if i = m + k - 1
                    update data structures with e_{m,n};
                if j = n + l - 1
                    update data structures with wires from e_{m,n} which
                      run in vertically in i;
                }
        }
    }
```

Figure 4.6: Feature Selection Algorithm

the permutation. These two procedures differ for the various permutations of compaction, wiring type and pass, and will be explained in detail in the sections below.

The section of the algorithm which generates the interference and wiring constraints is protected by the statement "if $i = m$ and $j = n$." This is done so that an element covering more than a single site will only be processed once, and so that the element will not be constrained to stay on the right of a guideline that runs through it. The statement, five statements below it, "if rigid compaction is being performed and $j = n$ and $i = m + k - 1$," performs a similar check for the guideline to the right of the element. In the final block there are two "if" statements. The first, "if $i = m + k - 1$" ensures that an element is not added to the data structures unless the right most portion of it has already been processed. Similarly, "if $j = n + l - 1$" ensures that the wires are not added to the data structures more than once.

### 4.3.1 Feature Table Data Structure

The selection algorithm just discussed must be able to find all the other features which could interfere with a given feature. To do so efficiently, two special data structures are required. They are the *feature table* and the *interval grove*.

The *feature table* is a table which contains features from a given row that:

- have already been processed by the selection algorithm in Figure 4.6, and
- may possibly interfere with other features to the right, i.e., features which are about to be processed by the selection algorithm.

In other words, the feature table contains previously handled features which may interfere with features about to be handled by the selection algorithm A separate table is used for each row.

The feature table must support two operations efficiently. The first operation is retrieval of all features in the feature table, and the second is updating the table with new features. This latter operation includes deletion of older features which are hidden by new ones. One feature will hide another on a given layer when it contains that layer and is placed in a higher numbered column (assuming column-wise compaction). If a feature is not hidden, it is exposed. When all of a feature's layers have been hidden, it is impossible for

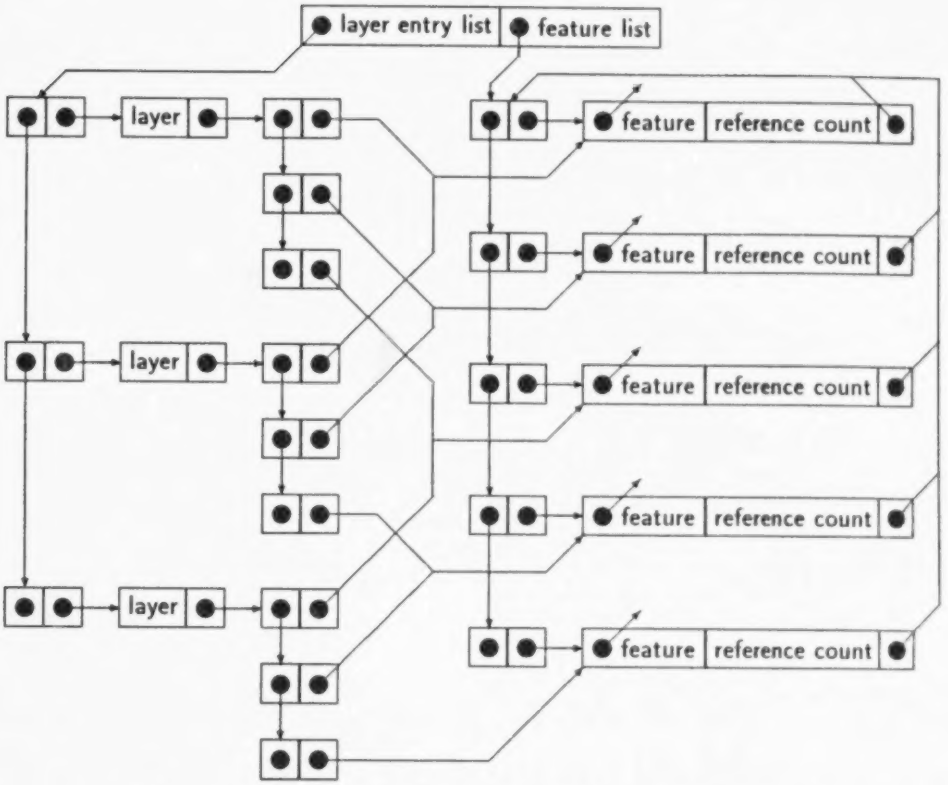



Figure 4.7: Example of a Feature Table Data Structure

it to interfere with features not yet encountered in the constraint generation process, thus dictating its removal from the feature table.

An example of a feature table data structure is illustrated in Figure 4.7. The structure principally comprises two lists, the *retrieval list*, used for fast retrieval, and the *layer entry list*, used to speed updating of the structure. The retrieval list can be seen on the right half of the figure, while the layer entry list can be seen on the left half. The layer entry list contains an entry for each process layer. Each of these entries contains a list of all the features in the table which contain that layer. The retrieval list simply contains all of the features in the table once and only once. Each feature has a reference counter and a pointer to the head of the retrieval list so that when it is deleted from the last list of a layer entry, it can remove itself from the retrieval list.

When the table is being updated with a new feature, it is put on the

retrieval list with a zero reference count. The update procedure then steps through the layer entry list. For each entry whose layer corresponds to a layer in the feature, the feature is placed on a list maintained for features exposed on that layer, and the reference count for that feature is incremented. Then, each other entry on the layer entry's list is checked to see if it is now hidden by the new feature. If it is, it is deleted from the layer entry's list, and its reference counter is decremented, and the feature is removed from the retrieval list if the counter has reached zero. When all the features must be retrieved, the retrieval list is used to quickly retrieve each feature once.

### 4.3.2 Rigid Straight Mode

Rigid compaction is distinguished from unconstrained compaction by the use of vertical lines to ensure that the column structure of the array is retained throughout compaction. The first line is fixed on the origin; subsequent lines are assigned to separate each column. These lines are constrained to keep their assigned order. When generating the constraints to prevent interference between elements, additional constraints are added to prevent each element from overlapping the lines on either side of its column or range of columns. This ensures that the column structure is retained, even when empty sites are included in the array.

Straight routing implicitly dictates the generation of equality constraints between connected elements. The fixed position of element terminals, and the requirement that the wires connecting the terminals remain straight, means that when the $x$ position of one of the elements is known, the other can be calculated. This was described in Section 4.2.2. These constraints are added to the tableau after the analysis for interference between a given element and others to its left has been concluded.

Efficient operation of these algorithms depends upon efficient search of the array to find potentially interfering features to the left of the current element. In all first pass algorithms, and in all rigid algorithms (i.e. excluding the second pass unconstrained algorithms), the feature table is used to keep track of these. These data structures must be initialized, updated and referred to by the constraint generation algorithms. Figure 4.6, the feature selection algorithm, shows the initialization of the feature tables. Figures 4.8 and 4.9 describe the first and second pass constraint generation algorithms for rigid straight mode respectively; the use of the feature table data structures to find potentially interfering features is shown there. After all constraints for a

GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$)

$A$: the array to be processed by *Amoeba*; $a_{i,j}$ is the site at column $i$, row $j$

$i$: the column index of the current site

$j$: the row index of the current site

$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$)

$D$: the set of all the data structures used to describe the features to the left of the current site; $d_h$ is the feature table for row $h$

```
{
for each feature table d_h ∈ D, where h ≥ n and h ≤ n + l − 1
    for each feature f in feature table d_h
        generate a constraint between f and e_{m,n};
}
```

GenerateWiringConstraints($A, i, j, e_{m,n}, D$)

$A$: the array processed by *Amoeba*

$i$: the column index of the current site

$j$: the row index of the current site

$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$)

$D$: the set of all the data structures used to describe the features to the left of the current site

```
{
for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
  e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
    {
    for each feature table d_h ∈ D, where h ≥ j + 1 and h ≤ q − 1
        for each feature f in feature table d_h
            generate a constraint between f and w;
    generate an equality constraint between e_{m,n} and e_{p,q};
    }
}
```

Figure 4.8: First Pass Rigid Straight Mode Algorithms

```
GenerateInterferenceConstraints(A, i, j, e_{m,n}, D)
A: the array to be processed by Amoeba; a_{i,j} is the site at column i, row j
i: the column index of the current site
j: the row index of the current site
e_{m,n}: the element occupying the current site (m and n are the row and
  column indices of the bottom left site occupied by e_{m,n})
D: the set of all the data structures used to describe the features to the left
  of the current site; d_h is the feature table for row h
    {
    for each feature table d_h ∈ D, which is closer than the maximum
      separation rule to e_{m,n}
        for each feature f in feature table d_h which is close enough to
          interfere with e_{m,n}
            generate a constraint between f and e_{m,n};
    }

GenerateWiringConstraints(A, i, j, e_{m,n}, D)
A: the array to be processed by Amoeba
i: the column index of the current site
j: the row index of the current site
e_{m,n}: the element occupying the current site (m and n are the row and
  column indices of the bottom left site occupied by e_{m,n})
D: the set of all the data structures used to describe the features to the left
  of the current site
    {
    for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
      e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
        {
        for each feature table d_h ∈ D, which is closer than the maximum
          separation rule to e_{m,n}
            for each feature f in feature table d_h
                generate a constraint between f and w;
        generate an equality constraint between e_{m,n} and e_{p,q};
        }
    }
```

Figure 4.9: Second Pass Rigid Straight Mode Algorithms

column have been generated, it is necessary to update each row's feature table with the elements and wires in this column. This is done before proceeding to the next column, as described in Figure 4.6.

During the first pass (Figure 4.8), when the selection algorithm is iterating over all elements in the array, GenerateInterferenceConstraints() checks each element for interference against any feature to its left in the same row or rows as itself. If interference is possible, an inequality constraint is generated. The inequality wiring constraints are generated when GenerateWiring-Constraints() looks to the left of the wire for all array sites it crosses, but not including the sites it originates in or terminates on. If the wire and another feature can interfere with each other, an inequality constraint is generated between the interfering feature and the original element of the wire. The equality wiring constraints (dictated by the choice of straight routing) are generated last.

The most significant difference between the first and second passes is the necessity of searching (in the second pass) outside the same row of the current element for potentially interfering elements. This is because the assumption used in the first pass, that the rows are separated by an infinite distance, is no longer valid. In the first pass, the $y$ positions of all the elements have been calculated and assigned. In the second pass, elements in different rows may be close enough (in the $y$ direction) to interfere with each other. Thus, any feature table which contains elements which could be close enough to interfere with the given element will have to be searched. This is as opposed to just searching the feature table for the rows occupied by the current element, as was sufficient during the first pass. This difference can be seen in the first statement of GenerateInterferenceConstraints() in Figure 4.8 and the second statement of GenerateInterferenceConstraints() in Figure 4.9, as well as the second statements in GenerateWiringConstraints in both Figures 4.8 and 4.9. In the second pass algorithm, a shortcut is used to speed the selection of features which might interfere with the element or wire under consideration. The maximum separation rule mentioned in Figure 4.9 is the largest separation required between any two layers in the process technology used for the array. Thus, if the bounding boxes of the two features being considered are separated by the maximum separation rule, or greater, they cannot possibly interfere with each other, and the detailed comparison of the two features on a layer-by-layer basis can be avoided.

### 4.3.3 Rigid Dogleg Mode

Doglegs are wires with jogs in them which allow the two elements to move independently of each other, thus removing the need for equality constraints between connected elements. However, space must be left in the layout for these doglegs. When generating constraints, it is not known if the wires will jog up or down, or right or left; therefore, space must be left for both possibilities. *Amoeba* treats this space as part of the element farthest from the array origin. For example, if the wire runs horizontally, the space is saved next to the element with the higher column index. Figure 4.10 illustrates horizontal wires and doglegs.

The algorithm in Figure 4.11 generates the new list of bounding boxes for an element and merges the doglegs into the bounding box list. Doglegs may jog to either the right or the left, and both cases must be handled. This is done by building two lists of bounding boxes, one for each direction, and then merging them. Since the merge produces the smallest bounding box which encloses both bounding boxes on a given layer, the new list of bounding boxes ensures that the worst case will be enforced. The algorithm will stack doglegs occurring on different, non-interfering layers on top of each other. Thus in Figure 4.10, the three wires illustrated must all be on mutually interfering layers.

*Amoeba* is not a compactor in the general sense. Dogleg routing (and unconstrained compaction) should not compromise *Amoeba's* ability to maintain the general array-like structure of its input. To this end, two special types of interference constraints have to be introduced when using dogleg

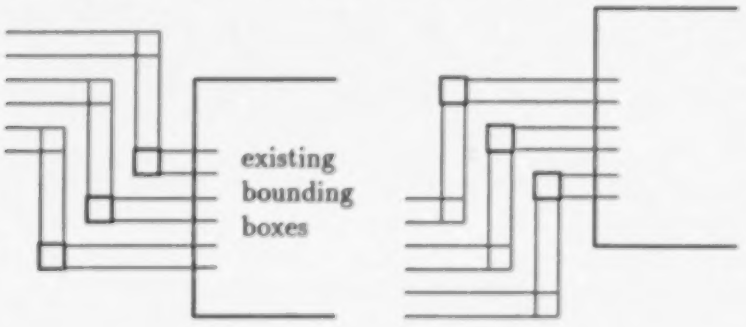



Figure 4.10: Doglegs and Bounding Boxes

```
CreateNewBoundingBoxList()
    {
    make a list of all horizontal wires terminating on this element;
    sort them in order from bottom-most to top-most;
    create a copy of the element's bounding boxes;
    for each wire, in order
        {
        create a minimum size square using the wire's layer, in its own
          coordinate space;
        using the interference constraint function, calculate the minimum
          separation between the square and the bounding box list copy;
        place the square opposite its wire's terminal at the minimum
          separation;
        merge the square and the bounding box list copy;
        }
    create a second copy of the element's bounding boxes;
    for each wire, in reverse order
        {
        create a minimum size square on the wire's layer;
        using the interference constraint function, calculate the minimum
          separation between the square and the bounding box list copy;
        place the square opposite its wire's terminal at the minimum
          separation;
        merge the square and the second bounding box list copy;
        }
    merge the two bounding box list copies;
    }
```

Figure 4.11: Algorithm for Including Doglegs in an Element's Bounding Boxes

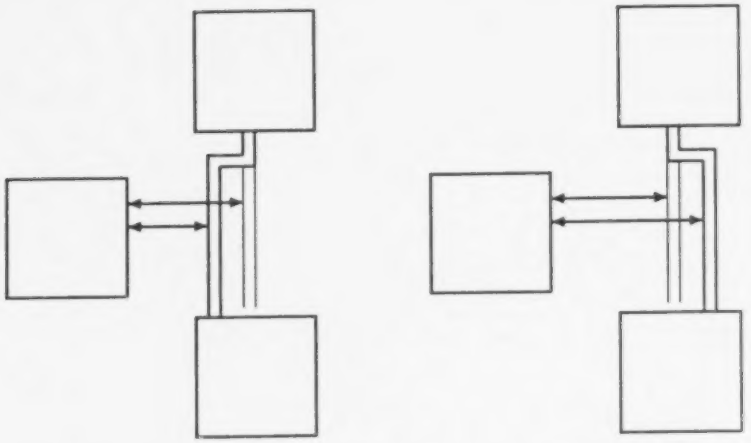
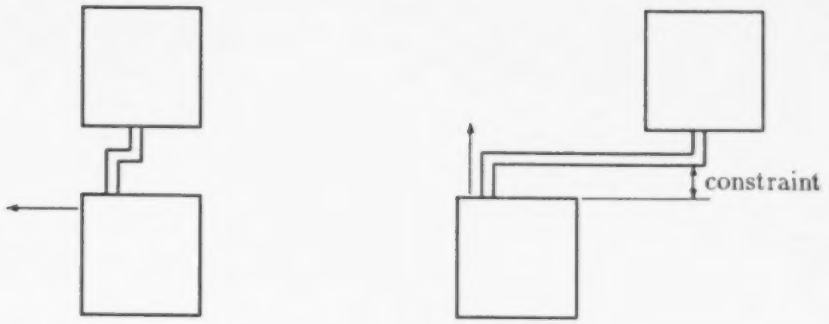


Figure 4.12: Constraint Pairs and Line of Sight Routing

routing.

A special line-of-sight constraint pair prevents an element from sliding between two other elements which are connected. This prevents the third element from blocking the "sight line" between the two connected elements. This sight line is usually a diagonal. Because its slope is unknown when *Amoeba* must constrain the potentially interfering element, an approximation to this constraint must be used. This is illustrated in Figure 4.12. Two interference constraints are used; one constraint, already required with straight routing, prevents the element from interfering with the wire. The other constraint prevents the element from interfering with the extension of the wire stub which connects the upper terminal to the horizontal jog. In the figure, this extension is shown with a lighter line weight than the wire itself. The extension is not generated as a part of the layout; it is only used during constraint generation. In effect, this constraint pair prevents the potentially interfering element from interfering with a rectangular region described by the wire's two terminals.

The second special interference constraint is designed to prevent two connected elements from being placed so close together that the dogleg interferes with the element with the lower index. This can happen if the lower indexed element slides far enough sideways that it cannot interfere with the connected element. The dogleg, however, extends out to the lower indexed element, and can interfere. This is illustrated in Figure 4.13. The method *Amoeba* nor-




First Pass: feature shifts left.

Second Pass: without the constraint, the lower feature could slide above the upper feature.

Figure 4.13: Preventing Interference with a Long Dogleg

mally uses to generate constraints will not protect the lower indexed element from interfering with this extended dogleg. *Amoeba* generates the special constraint by tracing each wire which terminates on the element *Amoeba* is currently examining, and which runs parallel to the axis of compaction, to its originating element. It then generates an ordinary interference constraint between the two elements as if they were adjacent.

The first pass algorithms are shown in Figure 4.14. In the first pass, the second special interference constraint is unnecessary: the feature table data structure and the feature selection algorithm guarantee that an equivalent constraint will be generated as part of the ordinary interference constraints. GenerateInterferenceConstraints() differs from the rigid compaction, straight routing case in three statements. The first two statements in this procedure have been added; they perform the merging of the doglegs into $e_{m,n}$ to produce $e'_{m,n}$. The final statement has been changed to use $e'_{m,n}$ instead of $e_{m,n}$, and if $f$ is a wire, two constraints are generated to preserve the "line-of-sight" between its originating and terminating elements. GenerateWiringConstraints() differs from the rigid compaction, straight wiring case in two ways. First, two constraints are generated to ensure that the terminals on either end of the wire remain "line-of-sight". Second, the equality wiring constraint is not generated, since the dogleg allows the originating and terminating elements to move relative to each other.

The second special interference constraint is required in the second pass.

GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$)

$A$: the array to be processed by *Amoeba*; $a_{i,j}$ is the site at column $i$, row $j$

$i$: the column index of the current site

$j$: the row index of the current site

$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$)

$D$: the set of all the data structures used to describe the features to the left of the current site; $d_h$ is the feature table for row $h$

```
{
create a copy of e_{m,n} called e'_{m,n};
merge doglegs on the left of e_{m,n} into e'_{m,n};
for each feature table d_h ∈ D, where h ≥ n and h ≤ n + l − 1
    for each feature f in feature table d_h
        generate a constraint or constraint pair between f and e'_{m,n};
}
```

GenerateWiringConstraints($A, i, j, e_{m,n}, D$)

$A$: the array to be processed by *Amoeba*

$i$: the column index of the current site

$j$: the row index of the current site

$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$)

$D$: the set of all the data structures used to describe the features to the left of the current site

```
{
for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
  e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
    for each feature table d_h ∈ D, where h ≥ j + 1 and h ≤ q − 1
        for each feature f in feature table d_h
            generate a constraint between f and e_{m,n} and between f
              and e_{p,q};
}
```

Figure 4.14: First Pass Rigid Dogleg Mode Algorithms

GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$)
$A$: the array to be processed by *Amoeba*; $a_{i,j}$ is the site at column $i$, row $j$
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site; $d_h$ is the feature table for row $h$

```
{
create a copy of e_{m,n} called e'_{m,n};
merge doglegs on the left and on the bottom of e_{m,n} into e'_{m,n};
for each feature table d_h ∈ D, which is closer than the maximum
  separation rule to e_{m,n}
    for each feature f in feature table d_h which is close enough to
      interfere with e'_{m,n}
        generate a constraint or constraint pair between f and e'_{m,n};
}
```

GenerateWiringConstraints($A, i, j, e_{m,n}, D$)
$A$: the array processed by *Amoeba*
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site

```
{
for each wire w which ends on e_{m,n} and starts on e_{p,q}, where p < m
    generate a constraint between e'_{m,n} and e_{p,q};
for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
  e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
    for each feature table d_h ∈ D, which is closer than the maximum
      separation rule to e_{m,n}
        for each feature f in feature table d_h
            generate a constraint between f and e_{m,n} and between f
              and e_{p,q};
}
```

Figure 4.15: Second Pass Rigid Dogleg Mode Algorithms

As well, both the doglegs below and to the left of each element must be included in $e_{m,n}$'s bounding boxes. This can be seen in the algorithms in Figure 4.15. Both GenerateInterferenceConstraints() and GenerateWiringConstraints() differ from their rigid, straight cousins in the second pass in the same way that the rigid, straight and rigid, dogleg first pass algorithms differ from each other. GenerateWiringConstraints() has one additional difference. The first two statements generate the second special interference constraint just described above.

### 4.3.4 Unconstrained Straight Mode

Only minor differences exist between *rigid straight mode* and *unconstrained straight mode*. In the feature selection algorithm in Figure 4.6, three pseudocode statements are protected by "if rigid compaction is being performed" statements. These sections control whether the guidelines which separate the rows and columns are generated. Without these guidelines, it is possible for an element assigned to one column to slide into another column. Unconstrained compaction is included in *Amoeba* to allow the designer the freedom to specify arrays which are most efficiently compacted in exactly this manner. The interference and wiring constraint algorithms used for unconstrained straight mode during the first pass are identical to those for rigid straight mode, as shown in Figure 4.8.

In the second pass, it is no longer safe to assume any row and column structure in the array. The first pass can move any element anywhere in the array where it is not blocked by another element. Therefore, the feature table data structure, used in all passes and modes previously discussed, and which rests upon the assumption of a fairly strict row and column structure, cannot be used. An *interval grove* data structure is used instead. This structure, described in detail in Section 4.3.5, keeps elements on a list sorted by their $y$ position (for columnwise compaction), instead of by row, as with the feature table structure. The elements are still assigned to rows and columns; the difficulty lies in the freedom of rows and columns to overlap.

The second pass algorithms, very similar to the second pass rigid straight mode algorithms, are shown in Figure 4.16. There are only three differences between the two sets of algorithms. The first statement in GenerateInterferenceConstraints() uses an interval grove instead of a set of dictionaries. The second statement in GenerateWiringConstraints() also has this difference. The unconstrained straight mode GenerateWiringConstraints() lacks a test

GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$)
$A$: the array to be processed by *Amoeba*; $a_{i,j}$ is the site at column $i$, row $j$
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site; in this case, the interval grove $g$

```
{
for each feature f in the interval grove g which is close enough to
  interfere with e_{m,n}
    generate a constraint between f and e_{m,n};
}
```

GenerateWiringConstraints($A, i, j, e_{m,n}, D$)
$A$: the array processed by *Amoeba*
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site

```
{
for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
  e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
    {
    for each feature f in interval grove g which is close enough to
      interfere with w
        generate a constraint between f and w;
    generate an equality constraint between e_{m,n} and e_{p,q};
    }
}
```

Figure 4.16: Second Pass Unconstrained Straight Mode Algorithms

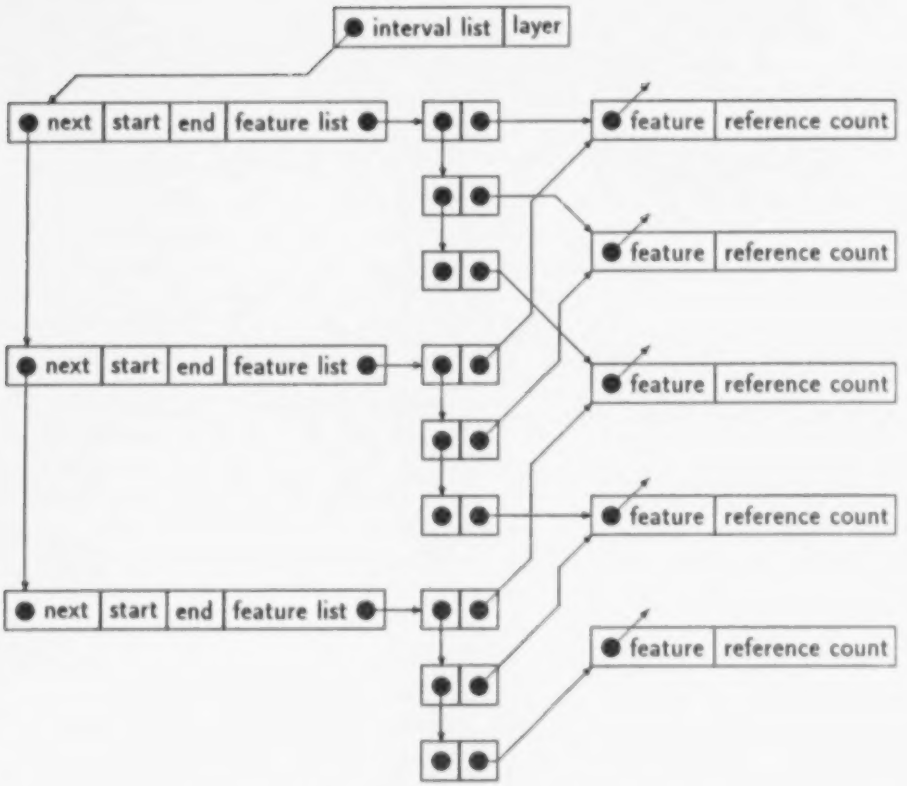



Figure 4.17: Interval List Data Structure

against the maximum separation rule, as this test cannot be applied when using an interval grove.

### 4.3.5 Interval Grove Data Structure

The *interval grove* is comprised of a number of instances of the *interval list data structure*. Each interval list keeps track of the interval of the $y$-axis occupied by each feature in itself and each process layer has its own interval list. Just as with the feature table data structure, all the features contained in the interval list could possibly interfere with unprocessed features to their right. Once a feature is entirely hidden in all intervals, it is deleted from the interval list.

The *interval grove data structure* was designed for fast lookup of an interval of interest, and fast insertion and deletion of features. The insertion and deletion of intervals is treated collectively as *updating*. An *interval grove* is a collection of *interval lists* for each process layer in the array. It gets its name (tongue in cheek) from an alternate implementation of the *interval list* — the *interval tree*. However, the greater difficulty in implementation, and the experience that the *interval list* implementation did not consume much processor time dictated retention of the simpler list implementation over that of the tree.

Figure 4.17 illustrates a typical interval list. The left side of the figure shows the *segment list*, while the right side shows a *feature collection*. A *segment* is defined here as an interval which contains the same features throughout its span, and which is as large as possible.

When *Amoeba* uses an *interval list* to look up all the features which cross a given interval, it searches the segment list until it finds a segment which intersects the interval. When all the features on that segment's feature list are returned to the procedure generating the constraints, then the next segment is examined. If this segment also intersects the interval, it is treated in the same manner. This process continues until the segment being examined is entirely beyond the interval. Then *Amoeba* stops searching, as all features have already been retrieved.

Updating the interval list is more complex. *Amoeba* searches the list for segments which overlap the interval covered by the feature with which it is updating the list with. When it finds a segment, it may have to split the segment in two, because the feature may only partly cover it. The portion that the new feature does not cover is unaffected and is left alone. The other portion is updated with the new feature. This involves examining the row or column of the features contained in the segment, and comparing the row or column to the new feature. For example, if *Amoeba* is performing columnwise compaction, and the new feature is in column five, all features ending in column four, or less, will be removed from the current segment, since the new feature will hide them. If the reference counter of the feature to be removed is zero after being decremented, it is deleted from the feature collection. If all the old features have been removed from the current segment, *Amoeba* will check the previous segment's feature list to discover if it also contains only the new feature; if so, the two segments are fused together. It is also possible that a new segment may have to be created if the new feature covers a hole in the segment list.

### 4.3.6 Unconstrained Dogleg Mode

*Unconstrained dogleg mode* combines features of both rigid dogleg and unconstrained straight modes. The differences between rigid and unconstrained compaction in the feature selection algorithm in Figure 4.6 were discussed in Section 4.3.4. As in the first passes of the rigid straight and unconstrained straight modes, the first passes of the rigid dogleg and unconstrained dogleg modes have identical interference and wiring constraint algorithms, which are shown in Figure 4.14.

In Figure 4.18, the hybrid nature of the *unconstrained dogleg mode* second pass algorithms can be seen, particularly if it is compared in contrast with Figures 4.15 and 4.16. Simply put, the *interval grove data structure* references replace the feature table references in the *rigid dogleg mode* algorithm.

GenerateInterferenceConstraints($A, i, j, e_{m,n}, D$)
$A$: the array to be processed by *Amoeba*; $a_{i,j}$ is the site at column $i$, row $j$
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site; in this case, the interval grove $g$

```
{
create a copy of e_{m,n} called e'_{m,n};
merge doglegs on the left and on the bottom of e_{m,n} into e'_{m,n};
for each feature f in the interval grove g which is close enough to
  interfere with e'_{m,n}
    generate a constraint between f and e'_{m,n};
}
```

GenerateWiringConstraints($A, i, j, e_{m,n}, D$)
$A$: the array processed by *Amoeba*
$i$: the column index of the current site
$j$: the row index of the current site
$e_{m,n}$: the element occupying the current site ($m$ and $n$ are the row and column indices of the bottom left site occupied by $e_{m,n}$) which is $k$ columns and $l$ rows in size
$D$: the set of all the data structures used to describe the features to the left of the current site

```
{
for each wire w which ends on e_{m,n} and starts on e_{p,q}, where p < m
    generate a constraint between e'_{m,n} and e_{p,q};
for each wire w which originates in site a_{i,j} and which connects e_{m,n} to
  e_{p,q}, where e_{p,q} occupies site a_{i,r} such that j < q
    for each feature f in interval grove g which is close enough to
      interfere with w
        generate a constraint between f and e_{m,n} and between f and
          e_{p,q};
}
```

Figure 4.18: Second Pass Unconstrained Dogleg Mode Algorithms

# Chapter 5

# Implementation and Results

This chapter is divided into two main sections: implementation and results. The implementation section briefly explains the approach taken to design and code the *Amoeba* computer program. The results section describes the timing and compaction results of three arrays which were used to test the effectiveness of *Amoeba*.

## 5.1 Implementation

*Amoeba* is written in C and was developed on Sparc (Sun 4) hardware running the SunOS variant of the Unix[1] operating system, although it is reasonably portable. *Amoeba* is comprised of 41 modules and 14000 lines of code. A further 5200 lines of code were written so that the modules could be tested. When running the example described in Section 5.2.1, the program occupied approximately 130000 bytes for its code, and another 525000 bytes for its data. It also used an undetermined amount of stack space.

This section covers three topics: programming, testing and efficiency. The programming section describes some general approaches taken toward coding the algorithms. The testing section describes a strategy and several techniques used to ensure that each module remained compliant with its specification through many versions of the program. The efficiency section describes some problems observed, and the changes that were made to speed up operation of *Amoeba*.

[1] Unix is a trademark of AT&T Bell Laboratories.

### 5.1.1 Programming

Parnas describes a software design technique called *information hiding* in [Par72]. This technique involves encapsulating every major design decision in its own module. The most common implication of this is that each data structure is encapsulated in a separate module. Information hiding is motivated by the desire to prevent changes caused by a change in one module from propagating out and affecting other modules. This technique was used extensively throughout *Amoeba*, and was very successful. Numerous times, data structures were substantially changed, yet only one module was affected.

### 5.1.2 Testing

During development, *Amoeba* evolved through many versions. As different modules were changed, it was necessary to ensure that all modules continued to operate correctly. Since most modules depend on other modules for various services, it was necessary to test each module which depended on an altered module, in order to verify that its behavior was still correct. To facilitate this, the module tests were included in the Unix "makefile" which is used as the input to the program *make*.

Each module had a test program written for it, which exercised all its routines and printed out the results. Once the module's operation had been verified, the test output was saved in a file. If a module on which the current module depended was later changed, *make* would rerun the test and compare the output from the new run to that which was saved. If they were identical, the current module would be marked as verified, and modules which depended on the current module could then be tested. The inter-module dependencies were entered directly into the makefile by the programmer. While not perfectly effective, this procedure caught many problems and isolated them at the module level, before the defective module was linked into an executable image of *Amoeba*.

One difficulty which presented itself early in the development of *Amoeba* is the printing of memory addresses contained in linked data structures. Every time any code in a module was changed, data structures would be placed in different locations in memory, and the comparison between the old and new output would fail. This could occur even if the current module was unchanged, but a module that the current module depended on had been changed. However, almost all the data structures used in *Amoeba* are dy-

namically allocated, using the standard C function call *malloc()*, and this suggested a partial solution. On the first call to *malloc()*, the address of the returned memory allocation is saved. Later, when the routines used to print out the data structures for testing were called by the test programs, the address of the first allocation was subtracted from all addresses before printing them. Since changes to the data structures were less common than changes to the module's code, and to other modules on which the current module depended, this method greatly reduced the number of automated test failures caused by inconsequential shifts of data structures in memory.

When a test did fail, several programs were used to speed up the task of isolating the cause of the failure. *diff*, a program supplied with Unix, compares two files, and prints out lines that are different. It can detect inserted or deleted lines and flag them as such. *diff* was very effective in speeding up the comparison of old and new output for differences, and was used extensively.

Sometimes *diff* would identify a very large number of lines as being changed, because a single item in a large number of data structures had changed, such as an address of another data structure. In this case *twin*, a program obtained from Usenet[2], proved useful. *twin* displays two files side by side, and highlights those characters which differ in the two files, typically by means of reverse video on the workstation screen. It does not identify inserted or deleted lines, so when these occur, and identical sections of output are out of step, they are flagged as different. Therefore both *diff* and *twin* were necessary, and complemented each others' functions.

Occasionally, as *Amoeba* developed, a change in a single module could affect the test output of many others. To speed the process of compiling and testing the different modules, a program called *testtool* was written. It is a simple program for running *make*, *diff* and *twin*, and accepting and saving new test output as correct. It has a graphical user interface, and was quite useful in the development effort for *Amoeba*. Unfortunately, it was not integrated with *make*, and certain pieces of data, which could have been obtained from the "makefile," had to be manually entered. An integrated environment for program development would benefit from including the functions of *make*, *diff*, *twin* and *testtool*.

[2]Usenet is a computer network which propagates computer programs and discussions on a wide variety of topics to its members, comprised of research and academic institutions, and businesses around the world.

### 5.1.3 Efficiency

As development progressed, it became apparent that certain parts of *Amoeba* were not as fast as desired or expected. This was caused in part by a design philosophy of "make it right first, make it fast second" which dictated simple solutions over fast, but complicated ones. Occasionally, the penalty for this approach was substantial, and algorithms had to be reworked. This section describes three of the larger changes that had to be made.

#### Pivoting

Initially, the tableau was implemented exactly as described in Section 4.1.4. Two arrays were allocated for the row and column headers, and the tableau elements were each contained in two singly-linked lists, each rooted in one of the headers. The tableau module implemented low level access functions for the tableau; however, pivoting was implemented in the simplex module, with the philosophy that it was a high-level operation, related more to the simplex algorithm than the sparse matrix implementation. A simple precursor to the gate matrix example described below in Section 5.2.1, contained seven columns and eleven rows and required 50 minutes to solve on a Sun 3/260 workstation. Several optimizations were required before solution time came down to less than two minutes. Two of these optimizations are described in the next two sections; two others are described in this section.

An execution profiler was used to analyze where *Amoeba* was spending its time. Profile reports of *Amoeba* revealed that the expensive operation was pivoting the tableau. The original pivot function, in the simplex module, treated the tableau as an array, and retrieved its elements via function calls to the tableau module. It used the row and column indices to retrieve an element. The first optimization was to speed up tableau element deletions through the use of doubly-linked lists instead of singly-linked lists in the sparse-matrix structure used for the tableau. Previously, the delete operation was given a pointer to an element to be deleted. It would look up the row and column numbers in the element, and then traverse both the row and column lists to remove the element from the lists. Now, when given an element to be deleted, it would simply use the back-pointers to remove the element from the list, thus making traversals unnecessary. This change made a noticeable improvement, but much less than the 50 to one desired to make *Amoeba* useful.

Soon after this was tried, a change was made to the overall design of

*Amoeba*. Rather than restrict the tableau module to low-level operations, it was decided to move the pivoting operation from the simplex module to the tableau module. The previous idea was to restrict the tableau module to elementary sparse-matrix behavior; however, this new approach effectively changed the tableau module from a simple sparse-matrix implementation that could be used in many different applications, to a tableau implementation, restricted to the simplex application.

The old pivoting operation treated the tableau as an array, and made no effort to optimize the traversals of the linked lists. The new operation, in the tableau module, pivoted the entire tableau in a single pass across all the parallel row lists. Thus, no link between elements was crossed more than once per pivot. This change was very successful, and was responsible for well over half the improvement in the running time of *Amoeba*.

### Memory Allocation

Until the problem in running time for medium sized arrays was found, all memory allocation and deallocation were done using the C functions *malloc()* and *free()*. These functions handle variably sized allocations. *malloc()* and *free* maintain a list of unallocated (free) blocks of memory. When memory must be allocated, *malloc* will traverse this free list, and when it finds a free block of sufficient size to accommodate the allocation request, carves the allocation out of this block. When memory is to be freed, *free()* takes the block, and traverses the free list to locate any immediately adjacent blocks. If any are found, they are merged with the block to be freed. These functions are fairly efficient for what they do; however, the tableau module must often allocate and deallocate tableau elements hundreds of thousands of times and the same small block of memory may be carved out of a larger block, and then merged again thousands of times.

A pair of functions was coded to handle the memory allocation and deallocation for the tableau module. The function, *tableauElementAllocate()*, would request allocation of a large block of memory from *malloc()*, and carve it up into individual tableau elements which were placed on the tableau element free list. Allocation from this list is extremely fast; move the free list pointer so it points to the second element on the list, and return the address of the first element. Deallocation is equally fast; link the deallocated element into the head of the list. If *tableauElementAllocate()* runs out of space on its free list, it simply requests a new allocation from *malloc()*. This new alloca-

tion is always twice the size of the previous allocation, making the number of calls to *malloc()* $O(\log n)$, where $n$ is the number of elements required. This was probably the second most effective optimization undertaken on *Amoeba*.

While the initial choice of *malloc()* may seem to be obviously poor in hindsight, *malloc()* is sufficiently fast to make only a trivial contribution to the running time of *Amoeba*, this in spite of the fact that it is used by the common singly-linked list code utilized for almost all the aggregate data structures in the program. Considering the high cost of using *malloc()* in the tableau module, it was surprising that profiling reports indicated that *malloc()* was quite efficient in all its other uses, and that it was not profitable to replace it in any other module. "Make it right first, make it fast second," seems to have served well here.

### Constraint Filtering

One final optimization will be described here. Late in the *Amoeba* development effort, it became apparent that certain combinations of input, compaction and routing options were producing many redundant constraints in the tableau. Since the size of the tableau is reflected in the running time for the simplex method, this was an obvious area for optimization.

A constraint filter was inserted between the module responsible for constraint generation and the tableau module. This was a simple task because of the clean interface between these two modules, and because constraints always consisted of one constant term and two variables with coefficients equal to one. The filter's data structure is a vector and each vector element points to a linked list of constraints whose lower subscript equals the vector's index. When a constraint is given to the filter, it changes the list element (constraint) on the appropriate list if it exists and if the new constraint is more restrictive, or it creates a new list element for the constraint if it does not exist. The filter collects constraints until constraint generation is complete; then it transfers them to the tableau.

This proved effective — the number of constraints to be included in the tableau was reduced by an average of 80.0%. Table 5.1 summarizes the performance of the constraint filter when used with the three examples discussed in the next section, with all combinations of modes.

| | Total | Average |
|---|---|---|
| Constraints Generated | 109271 | 2276 |
| Constraints Kept | 21872 | 455 |
| Difference | 87399 | 1820 |
| Percentage Difference | 80% | |

Table 5.1: Constraint Filter Performance

## 5.2 Results

*Amoeba* can be used to assemble two types of arrays — those comprised of minute detail, such as PLAs and gate matrices, and those comprised of larger structures, such as bit-slices and macrocells. Both types were tested here.

### 5.2.1 Gate Matrix Example

When Lopez and Law first described gate matrix layout, they used a half-adder bit slice as an example [LL80]. This array was shown in Figure 2.4 in Section 2.1.5. This bit slice is used here to demonstrate *Amoeba's* operation on arrays comprised of individual transistors and smaller pieces of geometry.

The example array has seven columns of polysilicon, and eleven rows, seven of which contain transistors. There are cells for $p$ transistors, $n$ transistors, polysilicon and diffusion contacts, and 3 micron squares of diffusion and metal which allow wires to be tacked together and to turn corners. All

| First Direction | Compaction | Routing | Height (microns) | Width (microns) | Time (seconds) |
|---|---|---|---|---|---|
| x | rigid | straight | 108 | 124 | 73 |
| y | rigid | straight | 107 | 148 | 101 |
| x | unconstrained | straight | 108 | 116 | 26 |
| y | unconstrained | straight | 107 | 118 | 21 |
| x | rigid | dogleg | 142 | 192 | 117 |
| y | rigid | dogleg | 141 | 209 | 200 |
| x | unconstrained | dogleg | 158 | 185 | 67 |
| y | unconstrained | dogleg | 137 | 218 | 75 |
| Manual assemble (Figure 2.4) | | | 110 | 122 | |

Table 5.2: Results for Gate Matrix Example

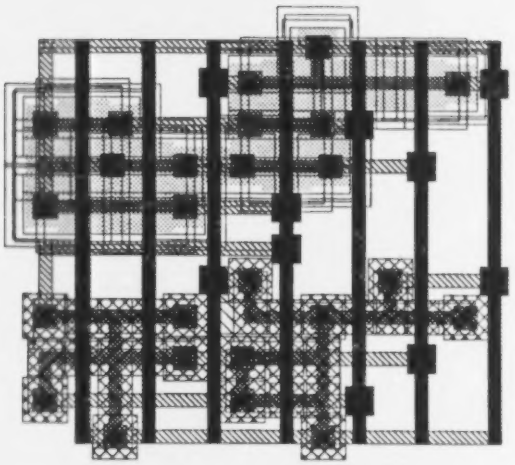


Figure 5.1: Gate Matrix Half Adder Layout Produced by *Amoeba* in Unconstrained Straight Mode

three examples use the Northern Telecom Electronics Ltd. CMOS3D process, which is a 3 micron, double metal, single polysilicon, single tub, $n$ substrate process. Appendix A describes the notation used in the diagrams of layout shown in this document. Figure 5.1 contains a diagram of the layout, generated by *Amoeba* using unconstrained straight mode. A complete listing of the array structure file can be found in Appendix C. The results for the array assemblies is given in Table 5.2. Figure 5.2 contains the same layout as Figure 5.1 enlarged and stripped of its routing so that only its cells are visible. The small dark-screened squares across the top and bottom of the figure are made of polysilicon and anchor the polysilicon columns. Where polysilicon appears over the diffusion regions (light-screened or cross-hatched), in the interior of the layout, transistors are formed. The 3 micron squares of metal (diagonal strips) or diffusion used to tack wires together can be seen beside many transistors in this figure.

The tables which describe the results of the examples used in this chapter all have the same format. The first column identifies the axis along which the array was first compacted. The second identifies the type of compaction (rigid or unconstrained), while the third identifies the type of routing (straight or dogleg). The fourth and fifth column give the resulting height and width of the assembled array in microns. Finally, the last column gives the elapsed time to run *Amoeba* on a Sun Sparcstation 2 workstation with 32 megabytes

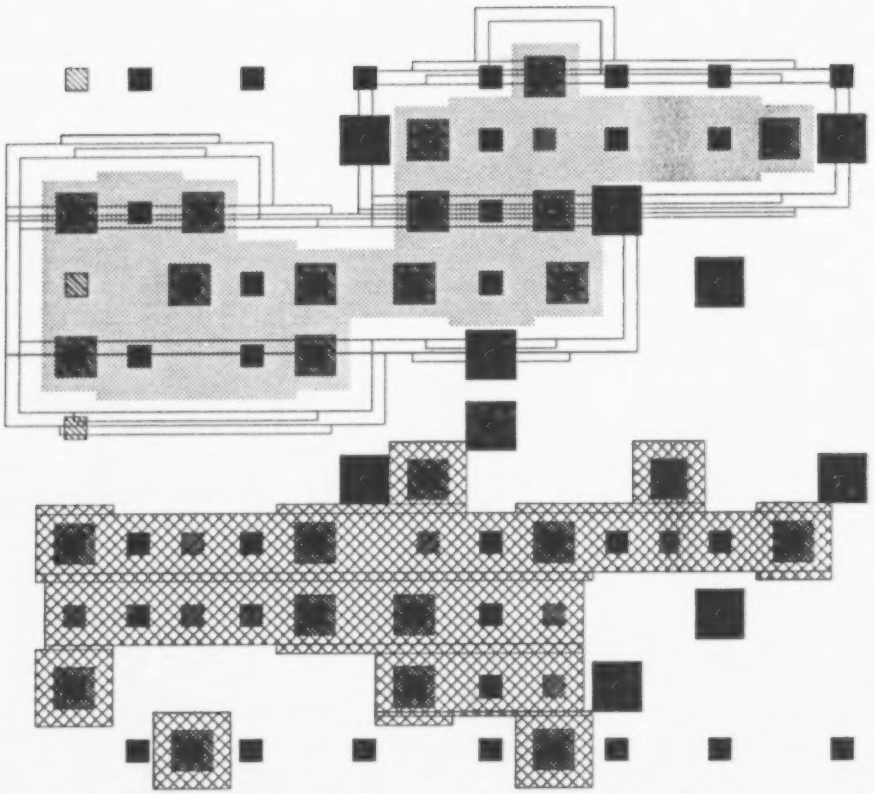


Figure 5.2: Gate Matrix Half Adder Layout Produced in Unconstrained Straight Mode, Stripped of its Wiring

of memory. The last line in table 5.2 gives the corresponding size of the gate matrix array, built using constant pitch, as described by Lopez and Law [LL80].

*Amoeba* performed well in this example. In less than thirty seconds, it can produce either of the unconstrained, straight arrays in less space than the hand-designed, constant pitch array. In these types of arrays, the unconstrained straight options will be the best performers, because they will allow the rows and columns to be pushed closer together without wasting space on the doglegs while the rigid straight options were almost as effective. In Figure 5.1, two contacts can be seen in the upper left corner, in between the first and second polysilicon columns. The contacts occupy the same interval on the $x$-axis although they originate from different columns. In the rigid

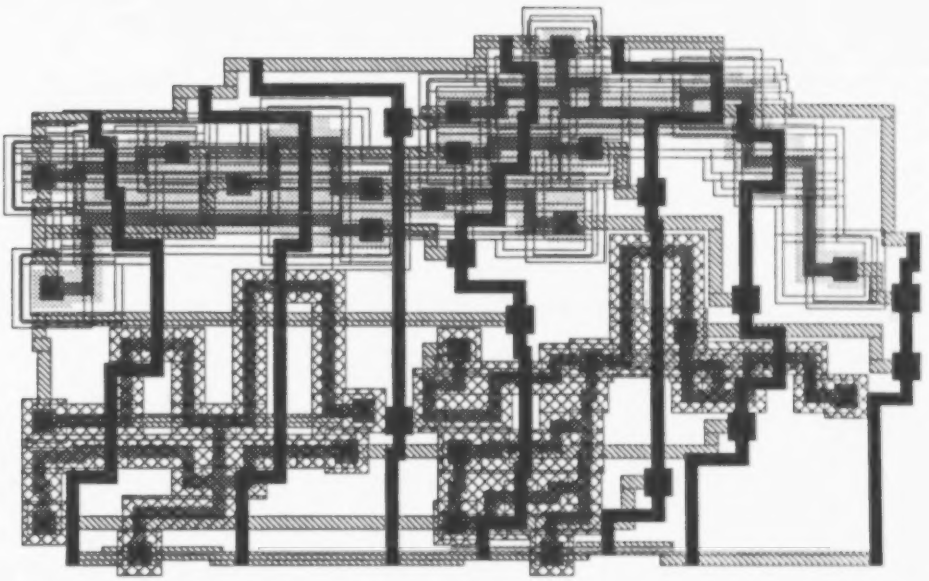


Figure 5.3: Gate Matrix Half Adder Layout Produced in Unconstrained Dogleg Mode

mode, these contacts cannot occupy the same interval, which means that the two polysilicon columns must be spaced further apart. When the dogleg cases are considered, it becomes apparent that the penalty for the doglegs far exceeds any space savings which could result from sliding the small elements around a few microns. Furthermore, the large amount of freedom that the doglegs allow *Amoeba* produces some very poor layouts. Figure 5.3 contains the layout for the $y$, unconstrained, dogleg case and shows some startling meanders connecting elements together. Both Figures 5.1 and 5.3 are shown to the same scale. Appendix C contains a listing of the array structure file for this example, as well as figures showing all the layouts produced from that file.

### 5.2.2 First ALU Example

Two examples of arrays comprised of larger scale elements are described in this and the following sections. Both are arithmetic logic units (ALUs) built out of bit slices. The first example is of an eight bit, twos-complement adder-subtracter, with two input registers and an output register. The control logic

| First Direction | Compaction | Routing | Height (microns) | Width (microns) | Time (seconds) |
|---|---|---|---|---|---|
| x | rigid | straight | 282 | 1427 | 14 |
| y | rigid | straight | 282 | 1427 | 10 |
| x | unconstrained | straight | 282 | 1427 | 25 |
| y | unconstrained | straight | 282 | 1427 | 13 |
| x | rigid | dogleg | 282 | 1553 | 17 |
| y | rigid | dogleg | 282 | 1553 | 13 |
| x | unconstrained | dogleg | 282 | 1553 | 33 |
| y | unconstrained | dogleg | 282 | 1553 | 17 |

Table 5.3: Results for First ALU Example

was not included in the example, but could be easily included in the array. The array has four rows and sixteen columns. The cells used in this example are comprised of a register cell, used in the bottom row, and this same register cell mirrored about the $y$-axis, used in the top row. An inverter cell used to facilitate subtraction is used in the second row, while a full-adder cell is used in the third row. All the cells are single bit-slices and were manually designed. The elements, except the input registers, are all two columns wide and can be seen in the second third and fourth rows. The results are given in Table 5.3. An example of the array, using the $x$, rigid and straight mode is shown in Figure 5.4.

All the straight routed arrays are identical, as equality constraints, caused by all the wiring, held the elements together in the same relative positions. The dogleg routed arrays are wider but not taller than the straight routed arrays. The extra width is accounted for by the spaces left for the doglegs between the elements. The arrays are not taller because most of the wiring parallel to the $y$-axis uses the metal-2 layer, and there are practically no features constructed on this layer in the elements. Thus, doglegs for the vertical wires lay over the elements, and no space was needed between the elements. The run times for these arrays are all under 35 seconds, which is reasonable.

### 5.2.3 Second ALU Example

The second ALU example is similar to the first, and uses many of the same cells. It is a unsigned, sixteen bit adder-multiplier. It has five rows and thirty-

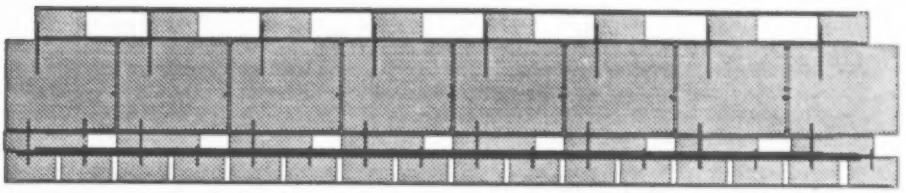


Figure 5.4: Rigid Straight Adder-Subtractor

two columns. As in the previous example, the control logic has been left out of the array. The ALU has two input registers, a thirty-two bit shift register which is used as the output register, and a multiplexor to select between an input register or the shift register. This allows the shift register to hold the multiplier when multiplying. Its cells are comprised of a register cell, two input, single output multiplexor cell, a full-adder cell, a shift-register cell, and this same shift-register cell, mirrored about its $y$-axis. Table 5.4 shows the results for this example, while Figure 5.5 shows a compacted ALU.

*Amoeba* produced an array with fatal errors in the x, rigid, dogleg run. Several wires crossed over each other, producing short circuits. This occurred because dogleg wiring is incompatible with wires running between two non-adjacent elements; there is no way to guarantee that the wires will not interfere in the general case, given the algorithms used by *Amoeba*.

Once again, the straight routing option produced the best layout, and the run times for the straight routing runs were all under two minutes.

In Figure 5.6, it can be seen that the current algorithms will produce

| First Direction | Compaction | Routing | Height (microns) | Width (microns) | Time (seconds) |
|---|---|---|---|---|---|
| x | rigid | straight | 370 | 2866 | 68 |
| y | rigid | straight | 370 | 2866 | 35 |
| x | unconstrained | straight | 370 | 2866 | 98 |
| y | unconstrained | straight | 370 | 2866 | 35 |
| x (see text) | rigid | dogleg | 371 | 3141 | 92 |
| y | rigid | dogleg | 371 | 3132 | 42 |
| x | unconstrained | dogleg | 371 | 3132 | 122 |
| y | unconstrained | dogleg | 371 | 3132 | 51 |

Table 5.4: Results for Second ALU Example

Figure 5.5: Rigid Straight Adder-Multiplier

"random" looking layouts when unconstrained compaction is selected and there are not enough interconnections between the elements to knit the array together; it would be preferable if this could be avoided. However, these are efficient layouts, and *Amoeba* has demonstrated its utility for assembling larger scale elements into arrays.

### 5.2.4 Discussion

The results of the gate matrix example, in Section 5.2.1, demonstrate that *Amoeba* can be quite useful for assembling programmable logic arrays, Weinberger arrays and gate matrices, using less area than do alternate methods of array assembly. *Amoeba* can simplify programming for the person who is designing a PLA generator, by hiding the details of tiling. And, it provides technology independence as well, further simplifying these generators. These same benefits are also available to the programmer who is developing an iterated structure generator.

All three examples failed to show that dogleg routing could make a significant contribution to tiling. In the course of testing *Amoeba*, it became apparent that it is not overly difficult to pitch match the terminals on the cells. The unconstrained compaction is able to provide most of the benefits that dogleg routing was supposed to provide. Furthermore, the doglegs take up extra space, and there are additional design problems when planning to use this feature. In particular, signals must not cross over elements, because *Amoeba* provides no control over where the wires will eventually be situated.

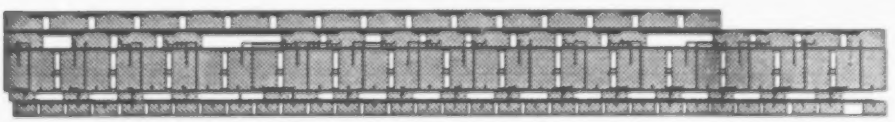


Figure 5.6: Example of "Random" Element Placement by *Amoeba*

# Chapter 6

# Conclusions

## 6.1 Summary of Results

*Amoeba* was intended to explore several new ideas for assembling array-like structures for the automated layout of VLSI. The primary idea was to replace the traditional tiling approach with an approach based on cells designed for minimum size. Connections are specified explicitly and this allows *Amoeba*, through the use of linear programming, to minimize the size of the array without violating any design rules.

*Amoeba*, for the most part, performs these tasks well. It can compact arrays comprised of small detail, such as PLAs and gate matrices, so that they are smaller than constant-pitch implementations. Arrays constructed of larger cells, such as ALUs, are also produced such that they are space-efficient.

In Section 5.2, the results of running *Amoeba* with several examples brought into question the utility of allowing doglegs in the wires connecting different elements. The doglegs occupy more space between the elements than straight wires, making the array larger. There are problems with wires which cross over one element enroute to another (a correct solution is not guaranteed by the current algorithms). And the motivation for allowing doglegs is not as urgent as originally thought: experience suggests that it is not difficult to ensure that the cells in a library used by *Amoeba* are pitch-matched. Thus, the minor compromises in the design of a pitch-matched library can be expected to be made up in the area savings realized by prohibiting doglegs.

## 6.2 Future Work

There are two kinds of changes to *Amoeba* which can be considered; these are enhancements to the current system and a wholesale change in the way the problem is cast. Three enhancements will be considered here, as well as a fundamentally different way of viewing this problem.

The first enhancement is a change in the ways in which the locations of terminals are specified to *Amoeba*. Rather than using a single point to describe the location, a rectangle could be used. Wires could be attached anywhere on one of its sides. This would be advantageous since signals in an element frequently do not simply terminate at a point, but could be made available anywhere along an internal wire which is used to connect internal cell structures together. This would allow the interconnecting wires to slide along a rectangle, giving more freedom for the positioning of the elements in an array.

Terminals are often placed near the perimeter of a cell to prevent interference from the interconnecting wires. Some terminals are not used every time a cell is; if a stub connects the terminal to the location for which the signal originates from in the interior of the cell, but the terminal is unused, it would be desirable to remove the stub, in order to reduce the capacitive load on the cell. The second enhancement would allow the stubs to be identified as such to *Amoeba*. They could then be removed automatically when the array is assembled, if they are unneeded.

The final enhancement would model the internal structure of an element in detail, instead of relying on the simplified bounding box structure currently used. This would allow some arrays to be further compacted. There are times, particularly in the first pass, when it is necessary to simplify the internal structure. However, the precise alignment of two elements is not known, the bounding box approach would still have to be used, as it is a fast method of calculating the worst case restrictions.

The most interesting change is that of the manner in which the array assembly problem is viewed. *Amoeba* allows the designer one choice with respect to doglegs: he uses them either on all wires, or on none. However, this is suboptimal. In some cases, certain doglegs could be eliminated, reducing the size of the array in one dimension, without affecting the size along the other array. The question is, which ones? If each wire which could be bent is assigned an integer variable, valued 1 if it is to be kept straight, or 0 if it is to be a dogleg, the original linear program is converted into a mixed integer

linear program. Unfortunately, mixed integer linear programming (MILP), is, in the general case, $\mathcal{NP}$-hard. Thus, a heuristic algorithm is necessary.

One possible heuristic is to calculate the critical-path through the graph formed by the constraints, and use this information to decide which wires will have doglegs and which will not. The program will calculate the longest path through a graph, as well as the range of element positions for non-critical elements. Once this has been done, the positions of noncritical elements could be assigned to minimize jogs or wiring length.

The following procedure may be an effective heuristic for this problem:

1. Calculate the constraints for a single direction. Calculate both the constraint if there are jogs to be accommodated, and if there are no jogs.

2. Determine the critical path using the "jogged" constraints. For all constraints on the critical path, switch to the "unjogged" constraints, and recalculate the critical path. Continue these iterations until the path is stable.

3. Assign the positions of the noncritical elements, attempting to eliminate jogs and minimize wiring length.

4. Repeat for the other direction.

How well this will work can only be known after programming and testing it on some examples.

# Bibliography


[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[BHA85] Cyrus S. Bamji, Charles E. Hauck, and Jonathan Allen. A design by example regular structure generator. In *Proc. of the 22nd Design Automation Conference*, pages 16–22, June, 1985.

[Can89] Canadian Microelectronics Corporation. *Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation*, GICIS Version 4.0, March, 1989.

[Cor85] Warren E. Cory. Layla: A VLSI layout language. In *Proc. of the 22th Design Automation Conference*, pages 245–251, June, 1985.

[Dan63] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[DB89] Mahesh H. Dodani and A. J. G. Babu. Karmarkar's projective method for linear programming: A computational appraisal. *Computers and Industrial Engineering*, 16(1):189–206, 1989.

[DN87] Srinivas Devadas and A. Richard Newton. Topological optimization of multiple-level array logic. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):915–941, November, 1987.

[Hil88] Dwight D. Hill. Alternative strategies for applying min-cut to VLSI placement. In *1988 IEEE International Conference on Computer Design ICCD-88*, pages 440–444, October, 1988.

[HW89a] Shuo Huang and Omar Wing. Improved gate matrix layout. *IEEE Transactions on Computer-Aided Design*, CAD-8(8):875–889, August, 1989.

[HW89b] Shuo Huang and Omar Wing. Gate matrix partitioning. *IEEE Transactions on Computer-Aided Design*, CAD-8(7):756–767, July, 1989.

[IO89] Mary Jane Irwin and Robert Michael Owens. A comparison of four two-dimensional gate matrix layout tools. In *Proc. of the 26th Design Automation Conference*, pages 698–701, June, 1989.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[KG85] A. J. Kessler and A. Ganesan. Standard cell VLSI design: A tutorial. *IEEE Circuits and Devices Magasine*, 1(1):17–34, January, 1985.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk* SSSR 244:5 (1979), 1093-1096, translated in *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[KMS84] Jin H. Kim, John McDermott, and Daniel P. Siewiorek. Exploiting domain knowledge in IC cell layout. *IEEE Design & Test*, 1(3):52–64, August 1984.

[KW85] Paul W. Kollaritsch and Neil H. E. Weste. Topologizer: An expert system translator of transistor connectivity to symbolic cell layout. *IEEE J. of Solid-State Circuits*, SC-20(3):799–804, June 1985.

[LL80] Alexander D. Lopez and Hung-Fai S. Law. A dense gate matrix layout method for MOS VLSI. *IEEE Transactions on Electron Devices*, ED-27(8):1671–1675, August, 1980.

[LM85] Hung-fai Stephen Law and John D. Mosby. An intelligent composition tool for regular and semi-regular VLSI structures. In *IEEE International Conference on Computer Aided Design ICCAD-85*, pages 169–171, November, 1985.

[LM87] Wen-Jeng Lue and L. P. McNamee. PLAY: Pattern-based symbolic layout; Part I: Transistor placement. In *Proc. of the 24th Design Automation Conference*, pages 659–665, June 1987.

[LNS+82] Richard J. Lipton, Stephen C. North, Robert Sedgewick, Jacobo Valdes, and Gopalakrishnan Vijayan. ALI: a procedural language to describe VLSI layouts. In *Proc. of the 19th Design Automation Conference*, pages 467–474, June, 1982.

[May86] Robert N. Mayo. Mocha chip: A system for the graphical design of VLSI module generators. In *IEEE International Conference on Computer Aided Design ICCAD-86*, pages 74–77, November, 1986.

[MBC83] T. G. Matheson, M. R. Buric, and C. Christensen. Embedding electrical and geometric constraints in hierarchical circuit-layout generators. In *IEEE International Conference on Computer Aided Design ICCAD-83*, pages 3–5, September, 1983.

[MK88] J. T. Mowchenko and D. Konrad. Amoeba: A generalized tiler. In *Proceedings of the Canadian Conference on Very Large Scale Integration CCVLSI 88*, pages 210–218, 1988.

[MN84] Grace H. Mah and A. Richard Newton. Panda: A PLA generator for multiply-folded PLAs. In *1984 IEEE International Conference on Computer-Aided Design ICCAD-84*, pages 122–124, November, 1984.

[MO83] Robert N. Mayo and John K. Ousterhout. Pictures with parentheses: Combining graphics and procedures in a VLSI layout tool. In *Proc. of the 20th Design Automation Conference*, pages 270–276, June, 1983.

[Mow87] J. T. Mowchenko. Algae and Amoeba: A pair of low-level layout tools. Technical report, University of Alberta, 1987.

[MSV83] Giovanni De Micheli and Alberto Sangiovanni-Vincentelli. Multiple constrained folding of programmable logic arrays: Theory and applications. *IEEE Transactions on Computer-Aided Design*, CAD-2(3):151–167, July, 1983.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December, 1972.

[PE84] Patrick A. D. Powell and Mohamed I. Elmasry. The ICEWATER language and interpreter. In *Proc. of the 21th Design Automation Conference*, pages 98–102, June, 1984.

[Poi89] Charles J. Poirier. Excellerator: Custom CMOS leaf cell layout generator. *IEEE Transactions on Computer-Aided Design*, CAD-8(7):744–755, July, 1989.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., 1982.

[RH85] Christopher Rowen and John L. Hennessy. Swami: A flexible logic implementation system. In *Proc. of the 22nd Design Automation Conference*, pages 169–175, June, 1985.

[SAK90] S. M. Sait and F. A. Al-Khulaiwi. Automatic Weinberger array synthesis from UAHPL description. *International Journal of Electronics*, 69(2):211–224, August, 1990.

[SAR90] S. M. Sait and Muhammad Abdul-Aziz Al-Rashed. Efficient algorithm for Weinberger array folding. *International Journal of Electronics*, 69(4):509–518, October, 1990.

[SMHO86] Walter S. Scott, Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout, editors. *1986 VLSI Tools: Still More Works by the Original Artists.* University of California at Berkeley, 1986.

[Wei67] Arnold Weinberger. Large scale integration of MOS complex logic: A layout method. *IEEE Journal of Solid-State Circuits*, SC-2(4):182–190, December, 1967.

[WL87] D. F. Wong and C. L. Liu. Array optimization for VLSI synthesis. In *Proc. of the 24th Design Automation Conference*, pages 537–543, June, 1987.

# Appendix A

# Legend for Diagrams of Mask Layout

The examples used in this research project were all constructed for the Northern Telecom Electronics Ltd. CMOS3DLM process. Figure A.1 explains the shadings and patterns used in the figures which illustrate these examples, and will remind the reader of some details of CMOS process technology. For a complete treatment, consult [Can89].

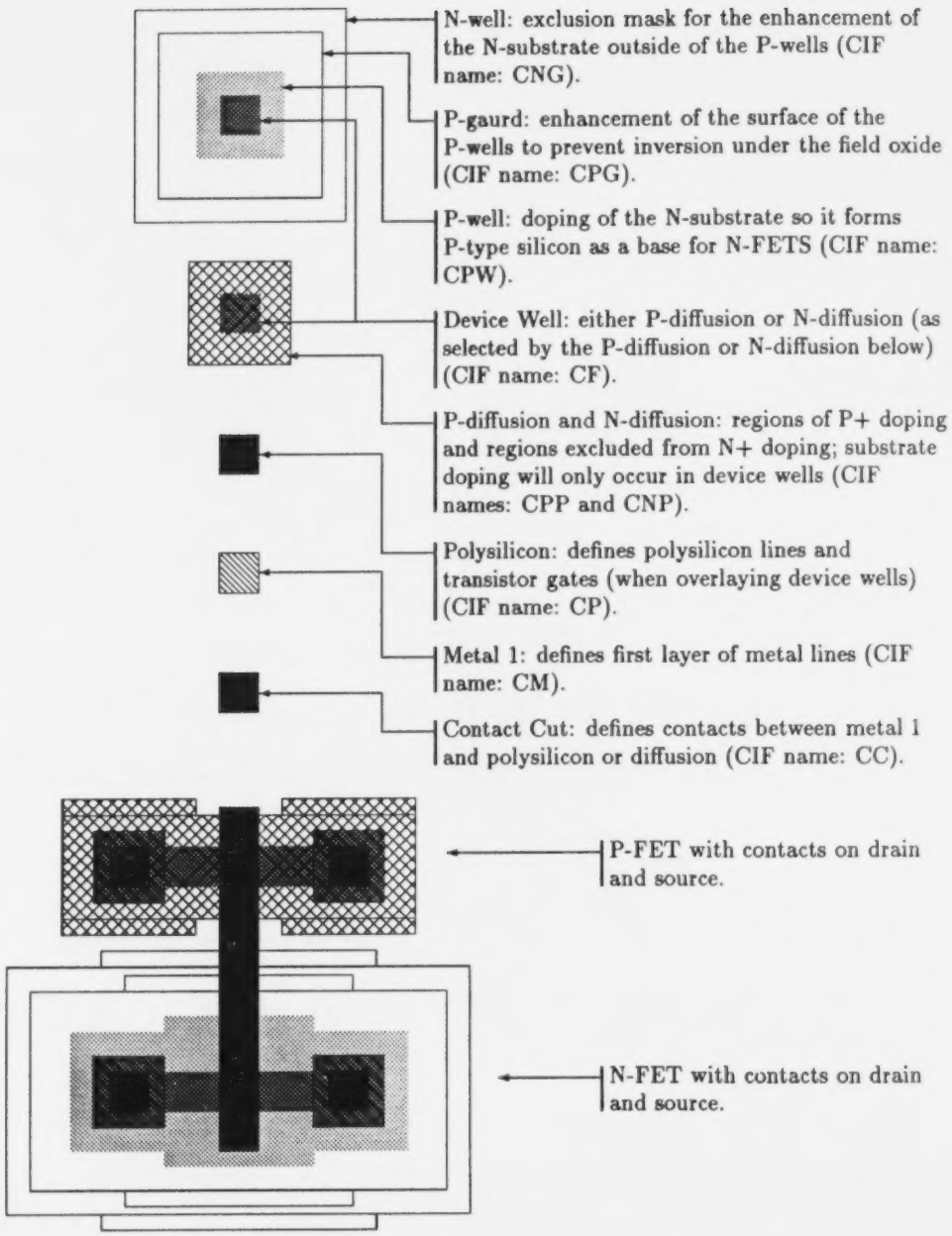

N-well: exclusion mask for the enhancement of the N-substrate outside of the P-wells (CIF name: CNG).
P-gaurd: enhancement of the surface of the P-wells to prevent inversion under the field oxide (CIF name: CPG).
P-well: doping of the N-substrate so it forms P-type silicon as a base for N-FETS (CIF name: CPW).
Device Well: either P-diffusion or N-diffusion (as selected by the P-diffusion or N-diffusion below) (CIF name: CF).
P-diffusion and N-diffusion: regions of P+ doping and regions excluded from N+ doping; substrate doping will only occur in device wells (CIF names: CPP and CNP).
Polysilicon: defines polysilicon lines and transistor gates (when overlaying device wells) (CIF name: CP).
Metal 1: defines first layer of metal lines (CIF name: CM).
Contact Cut: defines contacts between metal 1 and polysilicon or diffusion (CIF name: CC).
P-FET with contacts on drain and source.
N-FET with contacts on drain and source.

Figure A.1: Legend for Diagrams of Mask Layout

# Appendix B

# Meta-syntax for File Formats

In this document, file formats have been described with a variant of the Backus-Naur Form (BNF) for context-free grammars. This variant owes much to Section 4.2 in [ASU86].

1. Backus-Naur Form descriptions consist of a series of productions. A production describes the composition of a complex symbol. The symbol is placed on the left hand side, and its composition on the right. They are separated by an arrow ($\rightarrow$). Alternative compositions are separated by a bar (|).

2. The first production listed has, as its left-hand side, the start symbol. The start symbol is the symbol first expected by the parser, and usually represents the entire contents of the file being read.

3. Nonterminal symbols are composite symbols. They impose a hierarchical structure on the grammar. An example is the symbol *statement* which may represent a number of different structures, and each structure may consist of several terminal symbols. Nonterminal symbols are represented by a name in italics.

4. Terminal symbols are symbols which are treated, for parsing purposes, as a unit. They may be treated as either a literal or nonliteral symbol.

5. Nonliteral terminal symbols are expressed in bold type, such as **number**. A production is provided to characterize the symbol as a regular expression.

| | | |
|---|---|---|
| *BNF* | → | *production BNF* |
| | \| | *production* |
| *production* | → | *nonterminal* → *symbol-list* |
| | \| | *terminal* → *symbol-list* |
| *symbol* | → | *nonterminal* |
| | \| | *terminal* |
| | \| | **literal** (in typewriter face) |
| | \| | *regular-expression* |
| | \| | **hard-to-express-character** |
| *nonterminal* | → | **nonterminal-name** (in italics) |
| *terminal* | → | **terminal-name** (in boldface) |
| *regular-expression* | → | *subexpression regular-expression* |
| *subexpression* | → | **character** *operator* |
| | \| | *character-class operator* |
| | \| | **character** |
| | \| | *character-class* |
| *operator* | → | + |
| | \| | * |
| *character-class* | → | **[ character + ]** |
| | \| | **[ character - character ]** |
| **hard-to-express-character** | → | < name of character > |

Figure B.1: Backus-Naur Form of Backus-Naur Form

6. Literal terminal symbols are displayed exactly as they would be found in a file. The `typewriter` type style is used.

7. Regular expressions are used to describe terminal symbols which do not have a single, literal form. They are comprised of five notations. Literal characters are expressed with the typewriter type style as above. Subexpressions may be grouped with parentheses. Zero or more instances of an expression is denoted with an asterisk (*). One or more instances is denoted with a plus (+). Character classes are described with the notation [abc]. Ranges in a class can be expressed with a dash, such as [A-Za-z].

8. Certain hard to express characters may be written as <newline>.

A loose BNF for this BNF can be found in Figure B.1. While not a complete grammar, it should serve as an example for the preceding discussion.

# Appendix C

# Data

Many readers may want examine the arrays produced by *Amoeba*. In this appendix, all of the arrays produced for the three examples in Chapter 5 are shown, as are the input files which specify the arrays.

## C.1 Gate Matrix Example

This example is introduced in Section 5.2.1, where the array sizes and run times for *Amoeba* are presented. They are discussed in Section 5.2.4. The Figures C.1 through C.8 contain diagrams of the arrays produced by *Amoeba* and are all drawn to the same scale.

The text below is the input file for this example. Each polysilicon column is divided into three columns for *Amoeba*; the centre column contains the active regions of the transistors, and the left and right columns contain any contacts needed to allow signals to switch layers.

```
%C 1 0 vdd; p!p m1>m1
%C 2 0 pdcon; m1>m1 pd!pd
%C 4 0 vdd; p!p m1>m1
%C 7 0 vdd; p!p m1>m1
%C 10 0 vdd; p!p m1>m1
%C 11 0 pdcon; m1>m1 pd!pd
%C 13 0 vdd; p!p m1>m1
%C 16 0 vdd; p!p m1>m1
%C 19 0 vdd; p!p

%C 0 1 pdcon; pd!pd m1>m1
%C 9 1 pdcon; pd!pd pd>pd m1>m1
```

```
%C 10 1 ptran; p!p pd>pd
%C 11 1 pd; pd!pd
%C 13 1 pcon; p!p
%C 0 2 pd; pd>pd
%C 1 2 ptran; p!p pd>pd
%C 2 2 pd; pd>pd pd!pd
%C 4 2 ptran; p!p pd>pd
%C 5 2 pdcon; m1>m1
%C 9 2 pdcon; pd>pd m1>m1
%C 10 2 ptran; p!p pd>pd
%C 11 2 pd; pd!pd
%C 16 2 pcon; p!p
%C 0 3 pdcon; m1!m1 m1>m1 pd>pd
%C 1 3 ptran; p!p pd>pd
%C 2 3 pd; pd>pd
%C 4 3 ptran; p!p pd>pd
%C 5 3 pdcon
%C 9 3 pd; pd!pd pd>pd
%C 10 3 ptran; p!p pd>pd
%C 11 3 pdcon; pd>pd m1>m1
%C 13 3 ptran; p!p pd>pd
%C 14 3 pd; pd!pd pd>pd
%C 16 3 ptran; p!p pd>pd
%C 17 3 pdcon
%C 7 4 pcon; p!p m1>m1
%C 9 4 pdcon;
%C 14 4 pdcon; m1>m1
%C 19 4 pcon; p!p

%C 0 6 m1; m1>m1
%C 10 6 pcon; p!p
%C 0 7 ndcon; m1!m1 nd>nd
%C 1 7 ntran; p!p nd>nd
%C 4 7 ntran; p!p nd>nd
%C 5 7 ndcon; m1>m1
%C 10 7 pcon; p!p
%C 0 8 m1; m1!m1 m1>m1
%C 3 8 ndcon; nd>nd
%C 4 8 ntran; p!p nd>nd
%C 5 8 ndcon; m1>m1
%C 9 8 ndcon; nd>nd
%C 10 8 ntran; p!p nd>nd
%C 11 8 ndcon; m1>m1
%C 16 8 pcon; p!p
%C 0 9 ndcon; m1!m1 nd>nd
%C 1 9 ntran; p!p nd>nd
```

```
%C 2 9 ndcon; m1>m1
%C 9 9 ndcon; nd>nd
%C 10 9 ntran; p!p nd>nd
%C 11 9 ndcon; m1>m1
%C 13 9 pcon; p!p
%C 7 10 pcon; p!p m1>m1
%C 9 10 ndcon; nd>nd
%C 10 10 ntran; p!p nd>nd
%C 11 10 nd; nd>nd nd!nd
%C 13 10 ntran; p!p nd>nd
%C 16 10 ntran; p!p nd>nd
%C 17 10 ndcon; m1>m1
%C 19 10 pcon; p!p

%C 0 11 m1; m1>m1
%C 1 11 vss; m1>m1
%C 4 11 vss; m1>m1
%C 7 11 vss; m1>m1
%C 10 11 vss; m1>m1
%C 11 11 ndcon; m1>m1
%C 13 11 vss; m1>m1
%C 16 11 vss; m1>m1
%C 19 11 vss;
```

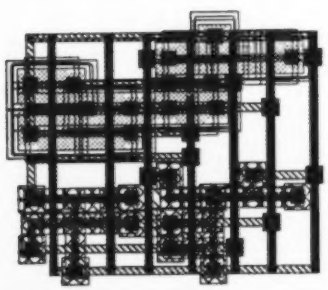


Figure C.1: Gate Matrix Example, X, Rigid, Straight

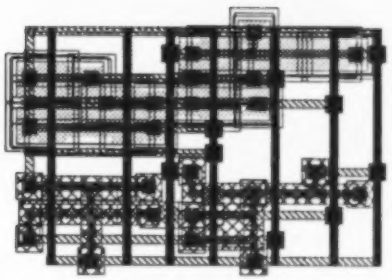


Figure C.2: Gate Matrix Example, Y, Rigid, Straight

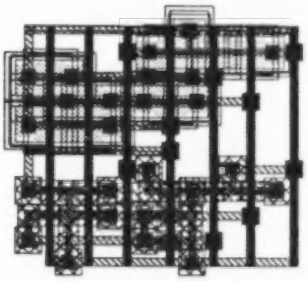


Figure C.3: Gate Matrix Example, X, Unconstrained, Straight

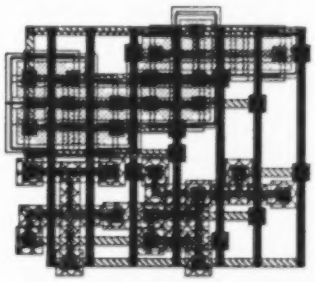


Figure C.4: Gate Matrix Example, Y, Unconstrained, Straight

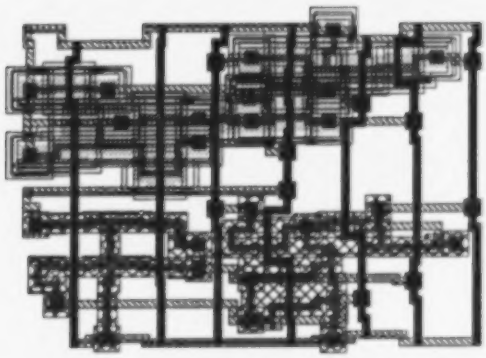


Figure C.5: Gate Matrix Example, X, Rigid, Dogleg

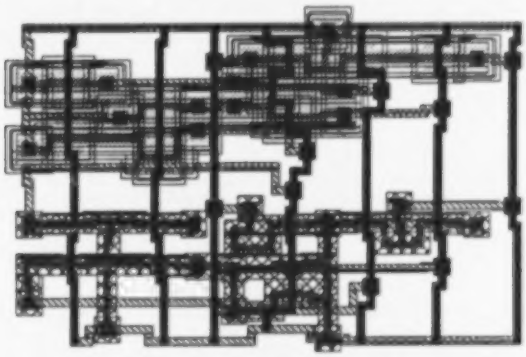


Figure C.6: Gate Matrix Example, Y, Rigid, Dogleg

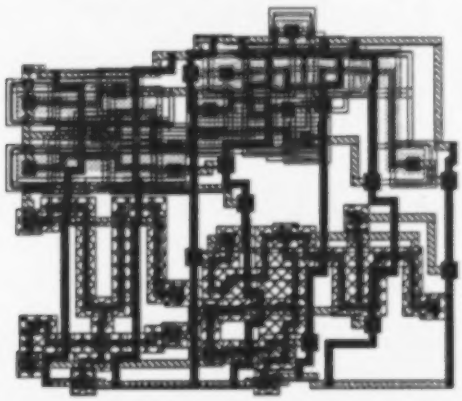


Figure C.7: Gate Matrix Example, X, Unconstrained, Dogleg

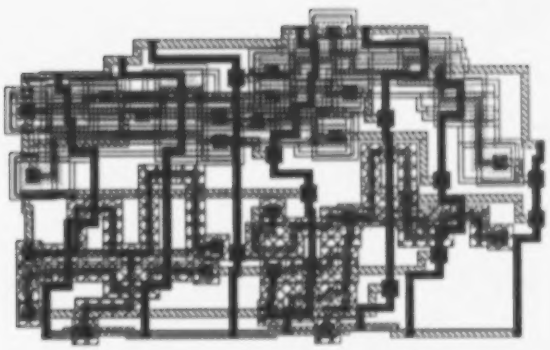


Figure C.8: Gate Matrix Example, Y, Unconstrained, Dogleg

## C.2 First ALU Example

This example is described in Section 5.2.2, where the results are summarized. The results are further discussed in Section 5.2.4. The Figures C.9 through C.16 contain diagrams of the arrays produced by *Amoeba*. They are all drawn to the same scale.

The text below is the input file for this example. Since large sections of the file are devoted to setting up rows of identical cells, with near identical wiring, repetitive statements have been replaced with ellipses. Interpolation of the omitted text should be obvious.

```
%C 1 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 2 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!y
%C 3 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 4 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!y

...

%C 15 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 16 1 register; out!y
%C 1 2 2 1 comp;
%O 2 2 comp; vdd>vdd vss>vss invert>invert notinvert>notinvert yprime!y
%C 3 2 2 1 comp;
%O 4 2 comp; vdd>vdd vss>vss invert>invert notinvert>notinvert yprime!y

...

%C 13 2 2 1 comp;
%O 14 2 comp; vdd>vdd vss>vss invert>invert notinvert>notinvert yprime!y
%C 15 2 2 1 comp;
%O 16 2 comp; yprime!y
%C 1 3 2 1 fadder; sum!in
%O 2 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout
%C 3 3 2 1 fadder; sum!in
%O 4 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout

...

%C 13 3 2 1 fadder; sum!in
%O 14 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout
%C 15 3 2 1 fadder; sum!in
%C 1 4 2 1 registerm;
%O 2 4 registerm; vdd>vdd vss>vss latch>latch notlatch>notlatch
%C 3 4 2 1 registerm;
%O 4 4 registerm; vdd>vdd vss>vss latch>latch notlatch>notlatch
```

...

```
%C 13 4 2 1 registerm;
%O 14 4 registerm; vdd>vdd vss>vss latch>latch notlatch>notlatch
%C 15 4 2 1 registerm;
```

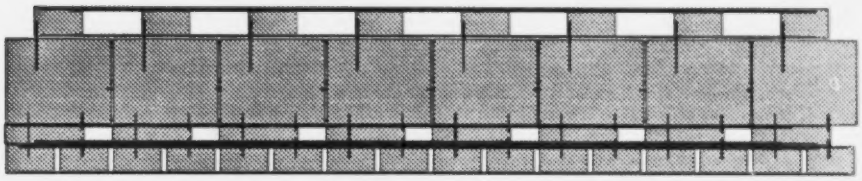


Figure C.9: First ALU Example, X, Rigid, Straight

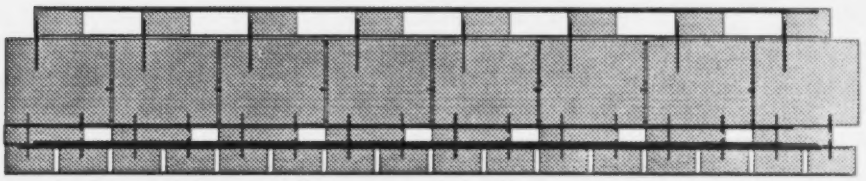


Figure C.10: First ALU Example, Y, Rigid, Straight

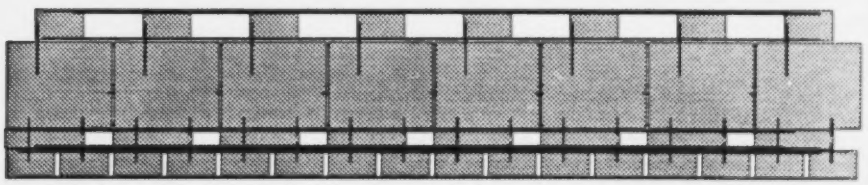


Figure C.11: First ALU Example, X, Unconstrained, Straight

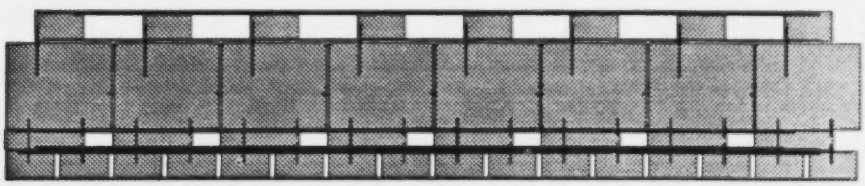


Figure C.12: First ALU Example, Y, Unconstrained, Straight

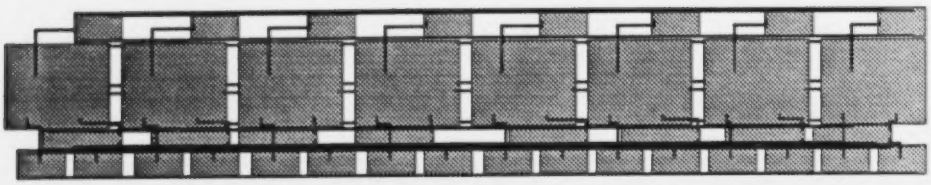


Figure C.13: First ALU Example, X, Rigid, Dogleg

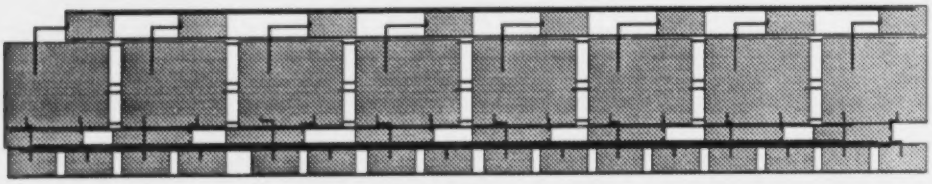


Figure C.14: First ALU Example, Y, Rigid, Dogleg

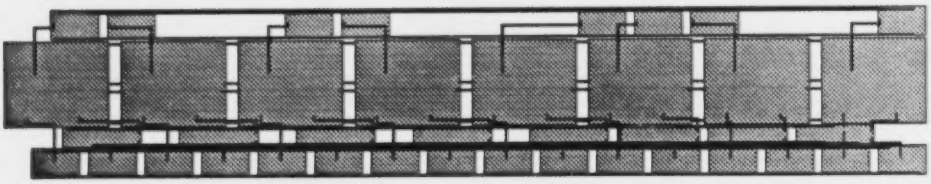


Figure C.15: First ALU Example, X, Unconstrained, Dogleg

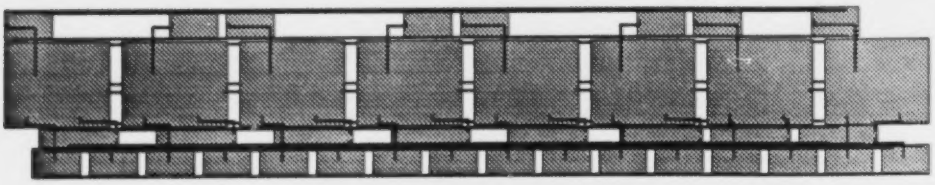


Figure C.16: First ALU Example, Y, Unconstrained, Dogleg

## C.3 Second ALU Example

This example is introduced in Section 5.2.3, where its results are presented. It is further discussed in Section 5.2.4. The Figures C.17 through C.24 contain diagrams of the arrays produced by *Amoeba*, and are all drawn to the same scale.

Several lines of the input file, below, were too long to fit on a single line. They have been truncated, and the truncated portion is shown on the following line, indented.

```
%C 1 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 2 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!y
%C 3 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 4 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!y
...
%C 29 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 30 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!y
%C 31 1 register; vdd>vdd vss>vss latch>latch notlatch>notlatch out!x
%C 32 1 register; out!y
%C 1 2 2 1 mux;
%O 2 2 mux; add>add mult>mult ymult!out yout!y
%C 3 2 2 1 mux;
%O 4 2 mux; add>add mult>mult ymult!out yout!y
...
%C 29 2 2 1 mux;
%O 30 2 mux; add>add mult>mult ymult!out yout!y
%C 31 2 2 1 mux;
%O 32 2 mux; ymult!out yout!y
%C 1 3 2 1 fadder;
%O 2 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout sum!in
%C 3 3 2 1 fadder;
%O 4 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout sum!in
...
%O 30 3 fadder; ruvdd>luvdd rgnd>lgnd rlvdd>llvdd cin>cout sum!in
%C 31 3 2 1 fadder;
%O 32 3 fadder; sum!in
%C 1 4 2 1 shiftreg
%O 2 4 shiftreg; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl
%C 3 4 2 1 shiftreg
%O 4 4 shiftreg; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl
```

```
...

%C 29 4 2 1 shiftreg
%O 30 4 shiftreg; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl
%C 31 4 2 1 shiftreg
%C 1 6 2 1 shiftregm
%O 2 6 shiftregm; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl
%C 3 6 2 1 shiftregm
%O 4 6 shiftregm; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl

...

%C 29 6 2 1 shiftregm
%O 30 6 shiftregm; vdd>vdd vss>vss latch>latch notlatch>notlatch
    shiftr>shiftl notshiftr>notshiftl
%C 31 6 2 1 shiftregm
```

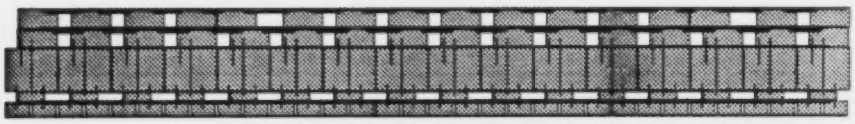


Figure C.17: Second ALU Example, X, Rigid, Straight

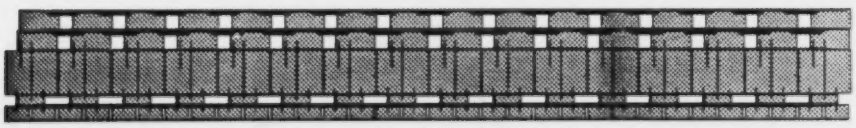


Figure C.18: Second ALU Example, Y, Rigid, Straight

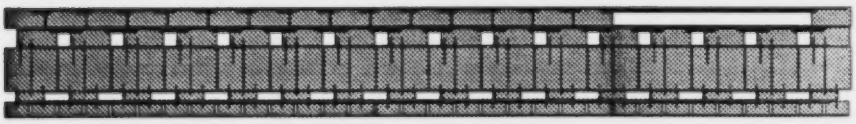


Figure C.19: Second ALU Example, X, Unconstrained, Straight

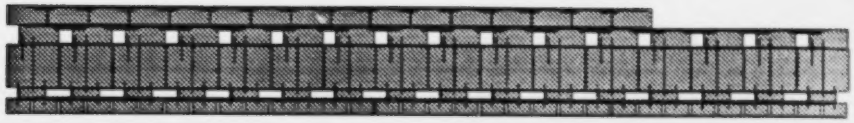


Figure C.20: Second ALU Example, Y, Unconstrained, Straight

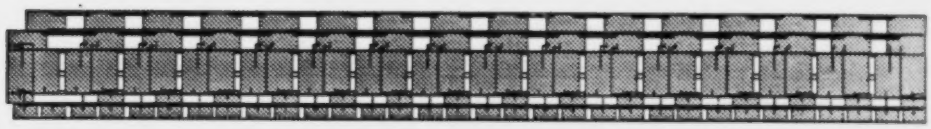


Figure C.21: Second ALU Example, X, Rigid, Dogleg

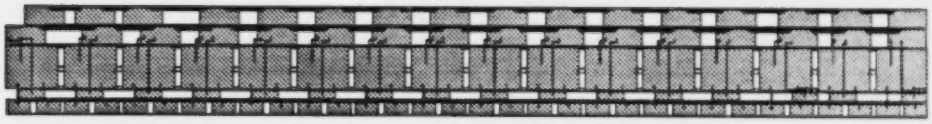


Figure C.22: Second ALU Example, Y, Rigid, Dogleg

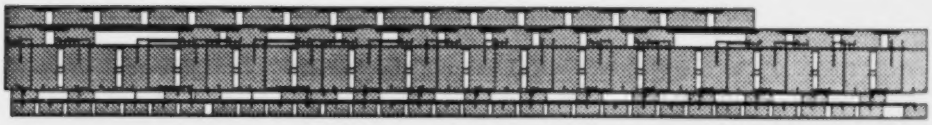


Figure C.23: Second ALU Example, X, Unconstrained, Dogleg

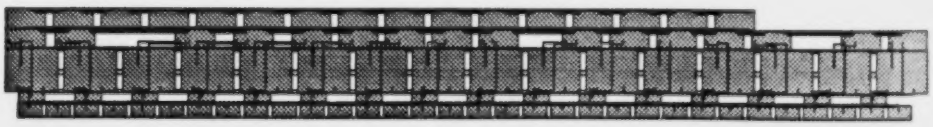


Figure C.24: Second ALU Example, Y, Unconstrained, Dogleg